A METRICS-BASED APPROACH
TO THE
AUTOMATED IDENTIFICATION
OF
OBJECT-ORIENTED
REUSABLE SOFTWARE COMPONENTS

by

LETHA HUGHES ETZKORN

A DISSERTATION

Submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy in

The Department of Computer Science

of

The School of Graduate Studies

of

The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

1997

DISSERTATION APPROVAL FORM

Submitted by Letha Hughes Etzkorn in partial fulfillment of the requirements for the degree of Doctor of Philosophy with a major in Computer Science.

Accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation committee:

_____ Committee Chair

(Date)

_____.

_____.

_____.

_____.

_____.

_____ Department Chair

_____ College Dean

_____ Graduate Dean

ABSTRACT

School of Graduate Studies

The University of Alabama in Huntsville

Degree Doctor of Philosophy College/Dept. Science/Computer Science.

Name of Candidate Letha Hughes Etzkorn .

Title A Metrics-Based Approach to the Automated Identification of Object-Oriented .

Reusable Software Components .

Software reuse has been a long term goal of software developers. This goal has been rather elusive, but the widespread use of the object-oriented paradigm and other innovations in software development paradigms have led to renewed interest in this topic.

This dissertation describes a new approach for the identification of reusable object-oriented legacy software components. The approach is concept-driven, knowledge-based and tool-assisted and uses heuristics to aid in the natural language understanding of comments and identifiers in object-oriented code. The approach uses a reusability quality factors hierarchy, where low-level object-oriented metrics are used to predict reusability quality factors.

The automation of this approach is provided by the Program Analysis Tool for Reuse (the PATRicia system), which is a software tool that aids in program understanding and metrics analysis. It consists of the Conceptual Hierarchy for Reuse including Semantics (CHRiS) tool, which uses a knowledge-base in the form of a conceptual-graph based semantic net in the natural language understanding of comments and identifiers, and the Metrics Analyzer tool which calculates object-oriented metrics that are then used to predict reusability metrics.

These tool-aided methods are shown to work well by identifying and qualifying reusable components in several real world graphical user interface packages. This approach demonstrates

that the identification of object-oriented legacy software components can be made significantly easier and more quantifiable than is possible using earlier techniques, which will aid greatly in promoting effective software reuse.

Abstract Approval: Committee Chair _____ _____.

(Date)

Department Chair _____ _____.

Graduate Dean _____ _____.

ACKNOWLEDGEMENTS

First, I want to express my gratitude to my advisor, Dr. Carl G. Davis, for his support throughout the course of this dissertation. I especially want to thank him for all the time he took to read and re-read this dissertation so as to make it as good as possible.

I also wish to thank the other members of my committee, Dr. Sajjan Shiva, Dr. Harry Delugach, Dr. Ashok Amin, Dr. James Hooper, and Dr. Maryam Asdjodi-Mohadjer for their advice and suggestions.

I want to thank the people who served as knowledgeable software evaluators, or experts, in the evaluation phases of my dissertation: Lisa Bowen, Brad Vinz, Dr. Minyoung Yun, LeeAnne Lewis, Janet Wolf, Tony Orme, and Randy Wolf. These people placed their considerable expertise at my disposal, and spent much time evaluating the **PATR**icia system. I also want to thank Bill Hughes and Karen Mallory for reading my dissertation and providing comments and suggestions to improve it.

I want to thank my husband, Dave, for his support throughout this dissertation. In addition to helping cook, clean, and run kids to dance class and ball practice, he also maintained three Linux systems for me and served as an extra software evaluator. I also want to thank my children, Patricia and Chris, for whom the **PATR**icia system and **CHR**i**S** tool are named, for letting Mama work on her dissertation when she needed to do so.

Finally, I want to thank my parents, William Evans Hughes and Margaret Watson Hughes, for their support in my educational endeavors throughout my life. My father passed away during the course of my Ph.D. work, so he did not live to see the end of it. I wish to dedicate this dissertation to him.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter I

INTRODUCTION

Software development in the United States alone was estimated to be a $74 billion business as of 1994, and has been increasing at a 9.6% annual rate over the past 10 years [105]. With the advent of new and innovative software controlled devices, software is gradually encroaching in almost every aspect of our lives. This creates pressure to develop more, better, and cheaper software. One obvious solution is to adapt and reuse software which has already been developed, rather than to develop new software. The ability of a 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 project, and hence avoid the costs of unnecessary redevelopment. While this goal is easy to state, there have been several barriers to software reuse. One of the more difficult barriers has been the lack of uniformity of products when software is developed using different paradigms. Combined with this is the lack of uniform standards which the products must meet, as well as the complexities of the software products being developed. All of these factors make it extremely difficult to effectively reuse large segments of previously developed products on new projects. Software reuse itself is a research area whose goal is to reduce or eliminate the barriers that prevent easy reuse of software products.

Software reuse is obviously a very worthwhile goal since it has been demonstrated to increase productivity, reduce costs, and improve software quality [59]. Much software reuse research in recent times has focused in several areas: supporting software reuse via improved managerial techniques, determining programming language mechanisms that would enhance software

reusability, establishing software reuse libraries, and their use during the software development process, and classification and retrieval techniques required for the use of software reuse libraries. Also, there has been some limited amount of research into developing automated tools to assist in identifying reusable subroutines or code fragments in legacy software systems. These tools have had varying degrees of success, but their development has helped identify the major hurdles to be overcome before software reuse becomes more commonplace.

Software reuse must also be considered in light of the major software development paradigm shift that has occurred over the last fifteen years. There has been a shift away from the traditional functional-decomposition paradigm in which a software system's capabilities were expressed in terms of functions and sub-functions. The shift has been to object-oriented development where what a system does is expressed in terms of interacting "objects" and classes of objects each of which provides a service. The collection of services that a software system provides defines the overall system capabilities. This shift has required many of the existing software reuse approaches that were useful in the functional-decomposition paradigm to be redesigned for the object-oriented paradigm.

One of the benefits claimed for software produced by an object-oriented paradigm is that it is inherently more reusable than functionally-oriented software. This is due in part to the fact that object interfaces are typically defined by "messages" and the internals of objects can be changed as long as the message form is preserved. The use of inheritance also provides a better means for identifying which parts of a software system depend upon each other. By using the object-oriented approach the ability to extract objects and classes and reuse them is enhanced, since their interfaces are well defined. Even with this inherent improvement many of the existing object-oriented legacy systems were not designed with reuse in mind, and their organization is such that reuse is still difficult, since documentation and interfaces were not designed to facilitate reuse.

Object-oriented software, however, due to the very aspects that make it desirable, tends to suffer from a wide scattering of the software that is required in order to perform even a fairly simple task. This scattering occurs since it is considered to be good object-oriented programming

style to write small message handlers [77], which results in an object-oriented system consisting of a large number of small modules. By the use of inheritance, a class may inherit one or more classes, with associated message handlers. Often these inherited classes are not defined locally. All of these aspects of object-oriented software tend to underline the need for good, semantically-based tools to aid in the understanding, and thus in the reuse of object-oriented software. Semantically-based tools provide a deeper understanding of the software than is provided by syntactically-based tools alone. The research described in this dissertation concentrates upon the development of an automated tool that uses semantic analysis to identify reusable components in object-oriented software.

In Chapter II a detailed description of software reuse issues, object-oriented software characteristics, program understanding theories, natural language understanding techniques, semantic networks, reusability metrics, and object-oriented metrics is provided. The overall research approach is described in Chapter III, the program understanding approach is described in chapter IV, and the reusability analysis approach is described in Chapter V. Chapter VI describes the Program Analysis Tool for Reuse (the **PATR**icia system), with its strengths and limitations. Chapter VII describes the results of the application of the **PATR**icia system to real C++ software. Together, Chapter VI and Chapter VII provide a proof of concept for the approaches described in Chapters IV and V. The conclusions of this research effort and future research directions are stated in Chapter VIII.

Appendix A provides details of the study of comments as a sublanguage. Appendix B provides supporting material for the study of the syntactic tagging of identifiers. Appendix C describes the results of the study of the application of the Sleator and Temperley natural language parser to comments. Appendix D provides the background of experts who were employed in the experiments that provided a validation of the operation of the **PATR**icia system. Appendices E and F contain the directions and questionnaires given to evaluators during the program understanding experiments that provided a validation of the program understanding approach followed in the **PATR**icia system. Appendix G contains the directions and questionnaires used in validating the reusability approach followed in the **PATR**icia system.

Chapter II

BACKGROUND

The development of a semantically-based approach for identifying software for reuse involves an understanding of a broad set of subdomains. These include reuse and preparing software for reuse, an analysis of the characteristics of object-oriented software, program understanding theories, natural language understanding, semantic networks and the use of conceptual graphs in natural language generation, reusability metrics and software metrics. Each of these subdomains will be described since they form the basis for the research described in the subsequent chapters.

2.1. The Software Reuse Problem

The term "software reuse" has been defined in several ways. Hooper and Chester [59] summarized nicely the following definitions of reuse:

1) "Reuse is an act of synthesizing a solution to a problem based on predefined solutions to subproblems." (Kang, 1987)

2) "Reuse is the process of implementing new software systems from pre-existing software." (AdaIC, 1990, quoting Cohen)

3) "Reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development and maintenance of that other system." (Biggerstaff and Perlis, 1989)

> 4) "Reuse is the process by which existing software work products (which may include not only source code, but also products such as documentation, designs, test data, tools, and specifications) are carried over and used in a new development effort, preferably with minimal modification." (Bollinger and Pfleeger, 1990)

The Global Protection Against Limited Strikes (GPALS) Software Reuse Strategy [54] states:

> "Software reuse is defined in many ways, but the simplest might be to say that reuse is the exploitation of commonality. It can also be defined as the process of implementing new software systems from pre-existing software."

As can be readily seen, the term is subject to a variety of interpretations. However, this has not prevented the development of software reuse strategies in recent years. Cavaliere [25][17][59] describes a successful reuse strategy involving COBOL programs at the Hartford Insurance Group. This reuse strategy had three goals: 1) to develop and implement a procedure for collecting and communicating reusable software, 2) to promote acceptance of reusable software as an effective productivity tool, and 3) to determine long-term support requirements for reusable software. Biggerstaff and Perlis [17][59] say that the ad-hoc approach used by Hartford Insurance provided good results, primarily due to management support and funding of reuse. This strategy was primarily management and process related, rather than product related. Specific information about designing reusable software, and how to choose reusable software from other less reusable software was only peripherally addressed.

Oskarsson [85][17] describes a successful reuse strategy for reusing public telephone exchange software. This software was formed of relatively large modules (more than 1000 executable statements each), in which no data is shared between modules, and there is no hierarchy of modules. Oskarsson identified several aspects of modules that aid in reusability. First, when a new module is added to a system, the identity of the module can be passed as a parameter to other modules, and there is no need to modify calling modules. Second, the abstraction of the module allows it to be used in very different environments. Strict locality of data and devices to a module encourages reusability by minimizing coupling between modules.

Oskarsson mentioned that the module concept of not sharing data resulted in a proliferation of interface modules for different devices. A study performed of the reuse process employed resulted in the development of a new system that consisted of 183 modules total with 80 modules being reused unchanged from a previous system. Additional modules were reused with modifications. Biggerstaff and Perlis [17] comment that the importance of this experience lies in the fact that the target environment makes development difficult and costly, since the real time software required must obey unusually strict memory and performance constraints, and yet reuse was still possible in this environment. This study was very specific to the module concept employed in the public telephone exchange software, although some of the insights gained are also applicable to object-oriented software.

Jette and Smith [61][17][59] describe a successful reuse strategy as well as the beneficial effect of object-oriented programming on reuse in their system HyperClass, a tool used at Schlumberger to develop oil well analysis tools. HyperClass is a user interface toolkit that encapsulates knowledge about user interfaces in terms of editors and editor parts [59]. This knowledge is used to implement application-specific extensions to user interfaces. Jette and Smith successfully focused on application domain reuse. They also investigated aspects of reuse using object-oriented programming, and emphasized that object-oriented programming reuses abstractions, and not just the code or the leaf nodes in a hierarchy [59]. They discuss the importance of designing a system with reuse in mind. This results in a commonality of interfaces, which improves reusability. They also mention that a weakness to their approach was the inability to determine the level of reusability of a class or set of classes. Jette and Smith mention that using HyperClass, interfaces built for applications require only a few days or weeks to build, which is an improvement over earlier interface design techniques. Biggerstaff and Perlis [17][59] consider HyperClass to be a baseline for others in the field of object-oriented reuse.

Selby [100][17] discussed a project that analyzed twenty-five software systems from a NASA production environment, ranging from 3000 to 112,000 lines of code. The amount of software either reused or modified from previous systems averaged 32 percent per project in this environment. Selby discussed total development effort per source line for modules that were

newly developed, extensively or slightly revised reused modules, and completely reused modules. New modules required most development effort per source line (1.089 tenths of hours/number of source lines), followed by extensively revised reused modules (0.758 tenths of hours/number of source lines), then slightly revised reused modules (0.601 tenths of hours/number of source lines). Completely reused modules required much less development effort than other modules (0.047 tenths of hours/number of source lines).

From these and many other case studies, software reuse has been demonstrated to increase productivity, reduce costs, and improve software quality [59]. However, preparing software for reuse can be costly. Hooper and Chester say [59]:

> "It costs more to prepare software for reuse than for a single use because of the extra effort required to generalize and to test the components, to document them well, and to classify and retain them for reuse."

Thus there are several financial considerations involved with software reuse. These include providing funding for training, additional personnel, and the generation of reusable components. Additionally, a mechanism for library storage and retrieval of reusable components must be provided, and time must be allocated in the product development schedule for retrieval, selection, and modification of reusable components.

Software reuse can be divided by application type into horizontal reuse and vertical reuse [59]. Horizontal reuse refers to the reuse of routines such as data structures and sorting algorithms, across a broad range of application areas, whereas vertical reuse refers to domain-specific components that can be reused again within a particular, narrow, and well-defined application domain. Vertical reuse requires a detailed domain analysis, which describes the common functions, data and relationships of systems within the domain. A single domain analysis can be used in multiple software development processes. Horizontal reuse is more common since it is easier to employ. However, greater leverage is possible through vertical reuse, but it is harder to accomplish because of the need for additional tools and techniques to support the reuse process. Development of these tools and techniques for object-oriented analysis and design approaches is central to the theme of this research.

2.1.1. Software Reuse Libraries

In order for a component to be easily reusable, it must be classified by the function or functions it performs and it should be stored for later retrieval in an on-line library. According to Ostertag and Hendler [86], the ability to reuse existing software requires four steps: definition, retrieval, adaptation, and incorporation. The *definition* step describes the component which needs to be constructed in terms of its functionality and relation to the rest of the environment. The *retrieval* step takes this description and retrieves from the software library a list of components with similar characteristics. One of these components is then selected. During the *adaptation* step the needed component is generated, usually by modifying the component that was selected from the library. The needed component is then *incorporated* in a new software package during development. Finally, new reusable software components are derived from the current software development project, and inserted into the software reuse library. The classification of a software component for easy later retrieval from a software library is an important part of the reuse procedure.

Approaches to software classification and retrieval have similarities with traditional library systems. Kim and Stohr [63] divide these approaches into the following categories: full text retrieval, keyword retrieval, the faceted approach, and enumerative retrieval. Novak [59] has a similar classification scheme.

The *full text retrieval* approach simply searches for all components within the software library that contain certain words specified by the user. The text searched can either be source code, or a short text description of a component stored in the on-line library. An example of this approach is that used by the CATALOG system from AT&T [59][53]. The full-text retrieval approach avoids the need for manual indexing, which is costly and can be inconsistent. However, as the library becomes larger, it becomes difficult to search and retrieve all the relevant resources, and to differentiate those from non-relevant resources [63]. Special Information Retrieval (IR) hardware can make this technique feasible for large libraries.

With the *keyword* approach, a set of keywords typically chosen from a standard list is attached to each component stored in the on-line software library. An example of this approach is that used by the NASA/Ames Research Center [63] [62]. A query for a keyword system consists of a list of keywords. This technique has the advantage of simplicity. However, the keyword list of all components in the library must be searched and then each component must be classified by a keyword list prior to insertion in a reuse library.

The *faceted* approach defines facets as the categories, or viewpoints appropriate to a particular domain [63]. A standard list of terms for each facet is stored by the system. Each item stored in a reuse library is described by the list of values it has in each facet. A system using the faceted approach is described by Jones and Prieto-Diaz [62], and by Prieto-Diaz and Freeman [93]. Advantages of the faceted method are that it allows multiple dimensions to be defined for relevant concepts, and standardizes the vocabulary for the concepts [63]. Each component must be classified by lists of values for each facet prior to insertion in the library.

The *enumerative retrieval* approach is similar to the traditional Dewey Decimal system. In this approach, a subject area is divided into hierarchically-arranged categories [63]. A category is attached to each component stored in the on-line software library. An example of this approach is that used by the International Mathematics and Scientific Library [60]. A query for a keyword system specifies a particular category. An inherent problem with enumerative classification schemes is the time required to search the hierarchical tree for the appropriate class [63].

One category of library retrieval mechanism that is mentioned by Novak [59] is knowledge-based library retrieval systems. An example of a knowledge-based system is that of Tarumi, Agusa, and Ohno [106][59]. Their system has a rule-based retrieval mechanism based on user inputs of object names, attributes, relations, and operations. Their approach is a mixture of formal and informal methods. Ostertag and Hendler [86][59] employ a frame system with multiple inheritance to represent the software library, and procedures used to find reuse candidates are based on artificial intelligence search techniques such as the A^* search algorithm. These knowledge-based systems are powerful, but can be slow, and the knowledge-based search must be performed any time a library search is required.

The work described in this dissertation will form a solid basis for the classification of reusable components. These knowledge-based techniques provide concepts embodied in the software, and quality metrics, which can be used as the basis of a keyword classification scheme, or that could be extended to other classification schemes.

2.1.2. Preparation of Legacy Code for Reuse

The generation of reusable components can be costly in terms of personnel and development time, and the classification of such components for reuse can also be expensive. The preparation of code components for reuse can be much harder if reusability was not considered when the code components were originally designed and implemented. Poulin and Caruso [90] say:

> "Failing to plan for reuse is the norm in traditional software development. In general, a software product is an original work except for the informal consideration of existing software for use in the new application."

This lack of planning for reuse has left immense bodies of existing software that has not been categorized or documented for later reuse. Dunn and Knight [39] say:

> "One often hears the comment that reusable components have to be designed rather that discovered, but to a software development manager with access to a large inventory of existing software, such a statement is both intellectually and economically dissatisfying. Intuitively, it is reasonable to assume that of the several billion lines of source code that have been developed over the years, some significant subset ought to be reusable with little or no modification. Location of reusable parts within existing software systems would be of considerable benefit, however, only if the parts were truly useful and the cost of finding them was less than the cost of rebuilding them."

This situation leads to the notion that the automated identification of reusable components in software would be cost effective. Only recently has any research focused on automated reuse identification. Two examples of automated reuse identification are those provided by Dunn and Knight [39] and by Caldiera and Basili [24].

Dunn and Knight have developed the tool called CodeMiner [39]. CodeMiner assists the programmer in identifying parts of legacy C software systems that might be potential candidates for a reuse library. CodeMiner identifies potentially reusable software in three ways: 1) It identifies functions that are invoked multiple times from multiple sections of code, 2) It identifies functions that are loosely coupled. Four types of coupling are checked for in CodeMiner:

a) data coupling--the functions share data through their interfaces,

b) common coupling--the functions share global data,

c) external coupling--the functions share data with the outside world, and

d) control coupling--the functions share data items upon which control decisions (such as branching) are made.

3) it identifies functions and global data elements that can be grouped together to form abstract data types. This is done by grouping together functions that use the same global data elements.

Caldiera and Basili developed the CARE (Computer Aided Reuse Engineering) system for the identification and qualification of reusable components in Ada and ANSI C legacy software [24]. The CARE system splits the analysis of existing programs into two phases. The first phase identifies some candidate components, and packages them for possible independent reuse. An engineer with knowledge of the application domain in which the component was developed then analyzes each component to determine the service that component can provide. Finally, components are stored in a repository along with all the information that has been obtained about them.

The component identifier (the first phase) selects some tentative candidate components and applies a reusability attributes model to the components to determine reusability of the component. The tentative candidate components are at the level of a C function or an Ada subprogram. Components whose measurements are within the reusability model's range of acceptable values become candidate reusable components to be analyzed by the domain expert in the qualification phase. This phase is fully automated.

The reusability attributes model measures reusability by the use of three quality attributes: Usefulness, Quality, and Costs. A value for Usefulness is determined by the use of McCabe's

cyclomatic complexity[81], by a regularity measure, and by reuse frequency. The regularity measure compares the actual length of a component to the estimated length, as determined by Halstead's Software Science [56]. The reuse frequency is measured by comparing the number of calls to a function to the number of calls addressed to a class of components that are assumed to be reusable (since the components have in the past been reused). The attribute Quality consists of several quality sub-factors. However, only sub-factors whose calculation can be automated are implemented in the reusability attributes model. The implemented sub-factors are correctness and testability, which are predicted using volume and cyclomatic complexity measures. The attribute Costs also consists of several quality sub-factors. Those chosen for use in the reusability attributes model are Qualification, Packaging, and Modification, whose calculation can be automated. Qualification is predicted using Halstead's volume. Packaging and Modification are predicted by the use of volume and cyclomatic complexity measures.

The Component Qualifier (the second phase) is an interactive assistance tool for a software engineer. It provides a "specifier" that supports the construction of a formal specification to be associated with the component. It provides a "tester" that uses the formal specification to generate or to support user generation of a set of test cases for the component. Finally, the classifier assists the software engineer in the selection of an appropriate classification for the reusable component.

These tools accepted as input legacy software written in C, ANSI C, and Ada. Neither of these systems, however, has addressed automated identification of reusable components in object-oriented software, which will be a focus of this research.

2.2. Object-Oriented Software

Software developed using an object-oriented approach generally includes components with four main aspects: identity, classification, polymorphism, and inheritance [99][22][91].

Identity means that each object has its own individual name. Two objects with different names are different objects even if all their attribute values and the services they provide are the

same. Within a programming language, each object has a separate "handle" by which it can be referenced.

Classification means that objects with the same attributes (data structures) and behavior (methods, member functions, or operations) are instances of a "class." A "class" is an abstraction that describes properties important to an application. A "class" can serve as the type definition of an object.

Polymorphism means that operations (methods, or member functions) with the same name may behave differently on different classes**.** The burden of deciding which implementation of a particular method to use is shifted from the calling code to the class hierarchy. For example, in a program to handle the display and manipulation of geometric figures, calling the "draw" method on a circle object results in a different output than calling the "draw" method on a rectangle object, and yet the name of the method is the same. In non-object-oriented software, the code calling the draw procedure would have had to select between two or more differently named draw procedures.

A different form of polymorphism is parametric polymorphism**.** Using parametric polymorphism, an object can respond to similar commands in different ways. Which method of an object responds to a command is selected through the use of types as method parameters.

Inheritance is a one-way hierarchical relationship in which attributes and methods are shared between classes, from a parent class (or parents, in the case of multiple inheritance) to one or more child classes.

Dynamic binding is a characteristic of object-oriented software that is associated with polymorphism and inheritance**.** A procedure call associated with a polymorphic reference may depend on the dynamic type of that reference. It may not be certain ahead of time which particular inherited property of an object's ancestors will apply to it at runtime. Dynamic binding means that the code associated with a given procedure call is not known until runtime [2].

Encapsulation, or information hiding, is another important characteristic of object-oriented software. With encapsulation, implementation details of an object are hidden from all code external to that object. Only a well-defined public interface, that reflects the application of the

object, is available externally. Thus the implementation of an object can be changed without affecting the applications that use it.

One of the primary characteristics of object-oriented systems is that objects represent an "abstraction," or a useful model, of part of the application domain. In an accounting package, for example, an object could represent a particular savings account, and the class of which the object was an instance would represent savings accounts in general. The concept of abstraction allows focusing on the essential aspects of an entity. Thus, ideally, design and implementation decisions are not made before the problem is understood.

The unit of primary reuse in object-oriented software is the class. A class is a construct for the implementation of an abstract data type (ADT). Due to the information hiding guidelines that are generally followed when designing object-oriented software, the interface of a class is an abstraction of a particular part of an application domain.

Verbatim reuse in object-oriented software is achieved by simply defining objects with the type of a reused class. Every time an instance of a class is created, reuse occurs. Leveraged reuse is achieved by the inheritance property of object-oriented software. If a class is almost, but not exactly what is needed, then defining a child class with some modifications allows the tailoring of the reused class in a way that does not introduce unwanted side-effects into the rest of the class [2]. Inheritance supports reuse across systems, and also supports extensibility within a single system. Korson and McGregor [66] give the following example: When a class Y inherits from a class X, Y usually has two parts, a derived part that was inherited from X, and an incremental part written for Y itself. In Y, a feature of X may be reimplemented, renamed, duplicated, voided, or have its visibility changed. Any new modifications to X, such as bug fixes, are automatically incorporated into Y. Abi-Akar [2] says that this allows a class to serve as the basis for many new definitions without propagating the errors of the original definitions throughout the system.

Another aspect of object-oriented software that aids in reusability is parametric polymorphism. Parametric polymorphism allows an object to refer to similar commands in different ways, by using types as method parameters. The most important advantage of parametric polymorphism is code sharing, which is a form of reuse.

Although the characteristics of object-oriented software result in code created in that paradigm being inherently more reusable than code created in the functional decomposition paradigm, still in the past ten years many object-oriented systems were designed and implemented without eventual reuse of the code being considered [83][84]. Thus the re-engineering of object-oriented legacy software for reuse is an important task.

2.3. Program Understanding

The identification of potentially reusable software components requires an understanding of the functionality of each software component. Program understanding or analysis in general includes any activity that uses dynamic or static methods to reveal the properties of existing programs. It most commonly refers to an examination of source code, without the use of any specification or execution information.

Several major hypotheses have been discussed in the literature in order to describe the program understanding process that is performed by a software engineer. These theories fall into two major categories: 1) bottom up program understanding, and 2) top down program understanding [38].

In bottom up program understanding, understanding begins at the syntactic level of source code. Statements of code are translated into a low-level semantic structure. Then familiar sequences of statements are recognized and grouped into understanding "chunks." Larger and larger "chunks" are pieced together until an understanding of the overall program has been achieved.

In the top down approach to program understanding, code is first scanned broadly. Multiple hypotheses are formed immediately about the program based on the high-level information. Each hypothesis is verified or rejected by the inspection of source code or documentation. When necessary, new hypotheses are examined. As hypotheses are verified, they are considered to belong to the overall understanding of the program. Complete understanding has been achieved when all of the hypotheses have either been verified or rejected. The total of the verified

hypotheses comprise the understanding of the program. If no hypothesis is verified, then the program considered to be not understandable.

Each of these approaches requires a program to be viewed from different levels of detail. Ning [82] provides a good classification of levels-of-detail program views: 1) implementation level, 2) structure level, 3) function (or functionality) level, and 4) domain level. Understanding at the implementation level requires knowledge about the syntax and semantics of the coding language. Understanding at the structure level requires the ability to derive information such as data/control flow graphs, interprocedural calling relations, or structure charts, and includes how these structures interrelate. Understanding at the function level consists of relating the pieces of a program to the functions that they perform. Understanding at the domain level requires abstracting away the algorithmic nature of the function (or functionality) view level, and relating the program to the requirements of the domain. Ning discussed the following example: in the context of "student record keeping," a program understood at the functional level as "computing average by summing up its inputs divided by the number of inputs" can be interpreted at the domain level as a "grade point average computation routine." Program understanding can be considered to be the process of moving from one program view to another, and this can occur in either a top down or bottom up manner.

The levels of detail required by different program views require different information. According to Brooks [23][38], the ability of a programmer to comprehend a program depends upon that programmer's programming knowledge, domain knowledge, and comprehension strategies. Brooks mentions that domain knowledge is crucial to the formation of the top level hypothesis. Thus, programming knowledge is related to implementation level, structure level, and function level understanding, while domain knowledge is only related to domain level understanding. Comprehension strategies comprise the process of moving from one level of detail program view to another.

Certain documentation aspects which improve program comprehension are described by Brooks [23][38]. Documentation which is internal to the program text includes: prologue comments; variable, structure, procedure, and label names, or identifiers; declarations or data

divisions; interline comments; and indentation, among others. Documentation which is external to the program text includes user manuals, flow charts, cross reference listings, or published descriptions of algorithms. Certain aspects of documentation are programming language specific, whereas others are domain specific. Belady and Lehman [12][38] contend that attempts to alleviate the problem of maintaining correct and complete documentation through the use of "self-documenting" high level programming languages have not been successful. Good documentation, including comments, is necessary not only to understand what the program is currently doing, but more importantly to serve as a source of information for reuse.

2.3.1. A Survey of Semantic Program Understanding Approaches and Systems

Using automated tools to aid in program understanding is a rich, ongoing area of research. Most of the earlier tools have been implementation and structure level understanding tools. Some of the more recent tools have used knowledge bases to provide functional (or functionality) level support. Very few tools provide domain level support, because it requires that application domain knowledge must be incorporated into the tool.

There are three large categories of program understanding tools, although some tools have aspects that cross categories. The *algorithmic approaches* annotate programs with formal specifications. These approaches rely on the user to annotate loops, and provide assistance only in proving the correctness of the annotations. The *knowledge-based approaches* rely on creation of a knowledge-base which can be matched to domain level as well as programming level concepts. These approaches typically annotate programs with English text descriptions. The *transformational approaches* are similar to the transformational paradigm of automatic program synthesis, but with the application direction of the transformation rules reversed.

One of the goals of this research is to produce a tool that automates the reuse process as much as possible. Thus the algorithmic approaches, which typically require user interaction, are not appealing in the context of this research. A source of difficulty with the transformational approaches is that the associated transformation knowledge-base tends to be very large. Most

importantly, neither of these approaches allow the identification of domain-level concepts. Thus a knowledge-based approach was chosen for use in this research.

The knowledge-based program understanding approaches can be divided into three major areas. The first approach, called the graph-parsing approach [95][96] translates a program into a flow graph, which shows both data and control flow. The domain base contains a library of graphical grammar-rule plans. The program's graphs are compared to the plans in the library, and recognition of program sub-parts becomes a graph-parsing problem. The second approach, called the heuristic concept-recognition approach [57][82], contains a knowledge-base of events, such as statement, control, inc-counter, dec-counter, bubble-sort, etc. An attempt is made to match the lowest level events to code statements. Then the low level events are combined to form high level events. The third approach, called the informal tokens approach [15][16][17][18][19], matches keywords taken from comments and identifiers to a knowledge-base. Recognized low-level concepts in the knowledge-base are combined to form higher level concepts. Some of the knowledge-based approaches use additional, structural information such as abstract syntax trees or call graphs to aid in identifying concepts.

Various program understanding tools are described below. The Computer Aided Reuse (CARE) system, which is the first system discussed, describes an algorithmic approach to program understanding, and the Cognitive Program Understander (CPU), the second system discussed, describes a transformational approach to program understanding. The other tools described employ various knowledge-based approaches. Most of these tools were primarily designed for use in program maintenance rather than in identification of reusable components (although they do have some application in that area as well). Thus, all of these tools are basically interactive tools that were intended to be a maintenance programmer's assistant; however, they do represent state-of-the-art in program understanding systems.

2.3.1.1. The CARE System/Caldiera and Basili

The previously mentioned CARE (Computer Aided Reuse Engineering) system also has a program understanding component. The Component Qualifier is an interactive assistance tool for a software engineer, that takes a formal, algorithmic approach to program understanding. It provides a specifier that supports the construction of a formal specification to be associated with a program component.

The formal specification program understanding approach used by the CARE tool does not address domain knowledge. Also, in some situations, a formal specification of a reusable component is not as desirable as a textual description of that component. An example of such a situation would be the case when some users of the library are not trained to understand formal specifications.

2.3.1.2. The Cognitive Program Understander (CPU)/ Letovsky

The Cognitive Program Understander [73][82] employs the reverse program transformation method. The reverse program transformation method is similar to automatic program synthesis, but with the direction of the transformation rules reversed. The Cognitive Program Understander uses lambda calculus as its program representation language. Lamba calculus is a language of functional expressions over a set of symbols. The lambda operator can create functions. Source programs must be translated into lambda calculus prior to the program transformation phase. Transformation rules are stored in a knowledge base. The lambda calculus representation of a source program is repeatedly transformed by matching the left hand side of a rule with part of the lambda calculus representation of the source program, then replacing it by the right hand side of the rule. A structure of program abstractions is then created. This is a bottom up program understanding approach.

This procedure suffers from various problems. First, each rule in the knowledge base can be matched only to a strict syntactic form. It has no method for providing a rule that can handle a

class of similar syntactic forms. Thus the knowledge base tends to be very large. Second, it is possible to have a "transformation cycle," where part of a program is transformed back into its original form. Finally, a transformation rule cannot match to non-adjacent program components, since the input conditions of the rule will not be satisfied. Thus program fragments that happen to be non-adjacent, although conceptually related, cannot be matched by a transformation rule.

2.3.1.3. The Programmer's Apprentice/ Rich, Waters, and Wills

The Programmer's Apprentice [95][96][110] is an intelligent programming assistant, rather than a fully automated tool, which was intended to provide interactive help to programmers in the program development and program maintenance domains. The approach is a bottom up program understanding approach.

A program in common LISP is first translated into a formal representation, or plan. The system for deriving formal representations is called the Plan Calculus. The Plan Calculus includes representations for control flow, data flow, and abstract data types. A plan is a directed, acyclic, graph in which nodes represent operations, and edges show the controls and data flows between the operations. A cliche library, in a knowledge base, contains a taxonomy of standard computational fragments and data structures which are also represented as plans. Cliche recognition identifies subgraphs in the program's graph and replaces them with more abstract operations to form an abstraction of the program. Cliche recognition is a graph parsing problem.

The Recognizer tool [96] finds all occurrences of a given set of cliches in a program, and builds a hierarchical description of the program in terms of the cliches it finds. The Recognizer forms a report by combining textual templates (in English) associated with recognized cliches, and filling in slots with identifiers taken from the program.

This approach requires a fairly large knowledge base for the understanding of even small programs, and thus can be difficult to apply to large systems. It is currently restricted to the analysis of small programs in common LISP. Also, domain level knowledge is not addressed [15][19].

2.3.1.4. The Program Analysis Tool (PAT)/ Harandi and Ning

The Program Analysis Tool (PAT) [57][82] is an interactive tool that takes a heuristic, concept-recognition approach to program understanding. It contains a knowledge-base of low level events, such as statement, control, increment-counter, decrement-counter, bubble sort, etc. The lower level events, called implementation level events, are combined to form the higher level events, called structure level events. Then structure level events are combined to form function (or functionality) level events.

At the lowest level, an attempt is made to match software statements versus the most primitive, implementation level events. The tool uses a technique, called Interval Calculus, that identifies control areas specified by line numbers surrounding a programming event, to perform the matching process. When all events have been identified, and the combination of events ceases, then the resulting high level events are considered to provide an understanding of the program under analysis. This is a bottom up program understanding approach.

Events, especially higher level events, have associated with them a structured English text description of the event, which are reported along with the event.

Programs input to PAT are small Pascal programs, with each program modified so that each statement is on a separate line with a unique line number. It has not been tested on large systems. Also, domain level understanding was not addressed by PAT.

2.3.1.5. Kozaczynski, Ning, and Engberts

Kozaczynski, Ning, and Engberts [67][68][69][70] use an approach that is primarily a heuristic, concept-recognition approach, but that also employs control flow and data dependency knowledge in the recognition of the most primitive level events. Some aspects of this tool are similar to those of PAT, the previously mentioned tool. In this approach, the program (in COBOL) is first parsed into abstract syntax trees. Each abstract syntax tree is augmented with control flow and data dependency knowledge. Then each abstract syntax tree is matched to a low

level concept. Higher level concepts are formed by combining low level concepts and meeting constraints due to control flow and data dependency. Plans in the domain base are stored in the following format:

Plan xx (higher level event)
consists of (lower level) event1, event2, event3
such that data dependency(event1, event2), control dependency(event 1, event3);

Understanding is complete when new concepts cease to be formed. This approach is basically a bottom up program understanding approach.

Kozaczynski, Ning, and Engbert's concept recognition tool is an interactive tool, with two windows: a program window, and a plan window. Text objects in the windows are directly connected with abstract syntax tree nodes in the knowledge base. The user can use all of the plans in the knowledge base to attempt to understand the program, or can select a subset of plans. After the concept recognition is complete, a dialog window appears that lists all the concept instances recognized, by name. When a concept in the dialog box is selected, the plan that contributes to its recognition is underlined in the plan window.

Kozaczynski, Ning, and Engbert's tool has been applied to large COBOL systems. However, domain knowledge is not currently addressed by the tool, although they do have plans for the addition of some domain knowledge. The tool is currently restricted to COBOL systems.

2.3.1.6. The LANTeRN System/Abd-El-Hafiz and Basili

Abd-El-Hafiz and Basili [1] developed a knowledge-based approach to the analysis of loops, as an important subset of program understanding**.** Their approach partook of ideas garnered from the areas of practical program decomposition, axiomatic program notation, and knowledge-based program understanding. Their approach mechanically documents programs by generating first order predicate logic annotations of loops.

Their process for loop analysis proceeds as follows: first, a loop is decomposed into fragments, called "events." Each event encapsulates the loop parts that are closely related with respect to data flow, and separates them from the rest of the loop**.** The resulting events are then analyzed using plans stored in a knowledge-base, to deduce their individual predicate logic annotations. Finally, the annotation of the whole loop is synthesized from the annotations of its events. This is a bottom up approach to program understanding.

The output of the LANTeRN (Loop ANalysis Tool for Recognizing Natural concepts) is the loop classification and loop events, along with the names of the plans that they match.

The LANTeRN tool currently operates only on a subset of Pascal, but should be extendable to larger systems. It is somewhat limited in scope, in that it is concerned only with loop analysis. Athough loops typically are one of the most difficult parts of a program to understand, loop analysis alone does not normally provide sufficient information upon which to base overall understanding of a program.

2.3.1.7. The DESIRE System/Biggerstaff

Biggerstaff [15][16][17][18][19] argues that a parsing oriented approach based on structural patterns of programming language features, which is the approach of all tools examined in this survey so far, is necessary but not sufficient for solving the general concept assignment problem. Parsing approaches return programming-oriented concepts such as searches, sorts, numerical integration, etc. However, human-oriented concepts such as acquire target and reserve airplane seat are not allied directly with algorithms. Thus, in order to extract these concepts it is necessary to extract informal information from the source code in terms of natural language tokens from comments and identifiers, as well as some heuristics related to occurrences of closely related concepts, and the overall pattern of relationships.

Biggerstaff's concept recognition prototype, the DM-TAO (Domain Model - The Adaptive Observer) from the DESIRE system (DEsign Information Recovery Environment), stores domain knowledge in the form of concepts within a semantic net. Natural language tokens, such as

identifiers and keywords from comments, are compared to low level domain concepts. Syntactic, lexical, and clustering clues are employed in initial low level concept recognition. The low level concepts are then used to form higher level concepts. Weights on the links of the semantic net are used to predict the accuracy of the concept recognition. If the weight is over a certain level, called the recognition threshold, then the concept is predicted to be present. The network weights must be empirically adjusted during performance in order to make the concept recognition accurate. This approach is basically a top down approach that is primarily concerned with domain knowledge; however, there are some bottom up aspects related to the operation of the semantic net.

The DM-TAO tool is a highly interactive tool with user selectable "slices" (interest sets), or highlighted areas, within the source code being the focus for program understanding. The output of the DM-TAO tool is a list of candidate concepts for the selected code. The user then can use structural information provided by various DESIRE tools to determine the accuracy of the assignment of the code in the slice to certain concepts. Based on this information, the user can change the weights of concept prediction. Thus, over time, knowledge is acquired that allows the system to improve its search and predictive power.

This approach has the advantage that it is easily extendable to large systems. However, the DM-TAO is still a research prototype, and has been applied only to small experiments.

2.4. Natural Language Processing

The understanding of various aspects of software documentation, such as comments is a natural language processing problem. This is integral to the approach described in Chapter III.

The processing of natural language can be divided into two primary subproblems. The first subproblem is the processing of written text, which requires lexical, syntactic, and semantic knowledge of the language, as well as real-world knowledge to provide a context for understanding the text. The second subproblem is the processing of spoken language, which requires all that is needed to process written text, plus additional knowledge about phonology and

enough added information to handle ambiguities that tend to arise in speech [97][3]. The natural language approach used in this dissertation will concentrate only on various aspects of the processing of written text.

The natural language understanding process can be roughly divided into the following subcategories [3][97]:

1) morphological analysis--individual words are analyzed into their components (such as roots and prefixes or subfixes).

2) syntactic analysis--linear sequences of words are transformed into structures that show how the words relate to each other.

3) semantic analysis--a mapping is made between syntactic structures and the problem domain. It concerns what words mean, and how these meanings combine in sentences to form sentence meanings.

4) discourse analysis--the meaning of an individual sentence may depend on the sentences that precede it, and may influence the meanings of the sentences that follow it. An example is pronoun interpretation.

5) pragmatic analysis--the interpretation of sentences based on current situation and usage.

Allen [3] gives an example to assist in distinguishing between syntax, semantics, and pragmatics. Consider the following sentences:

1) Language is one of the fundamental aspects of human behavior and is a crucial component of our lives.

2) Green frogs have large noses.

3) Green ideas have large noses.

4) Large have green ideas nose.

When considering each sentence as the first sentence in a textbook on natural language processing, sentence number one is syntactically, semantically, and pragmatically well-formed. Sentence number two is well-formed syntactically and semantically, but not pragmatically since

the subject matter has little to do with a textbook on natural language processing. Sentence number three is well-formed syntactically, but not semantically or pragmatically. Semantically, it is not possible for ideas to be green or to have noses. Finally, sentence four is not well-formed in any way, neither syntactically, semantically, nor pragmatically, since it does not obey the rules of English grammar.

The purpose of syntactic analysis is to build a structural description of a sentence. The goal of this process, which is called "parsing," is to define each of the words in the sentence by the type of word, and its usage in the sentence. Certain sentences may have several valid structural descriptions, or "parses." These sentences are termed "ambiguous" sentences.

Semantic analysis has two important tasks. First, it must map individual words into appropriate objects in the knowledge base. Second, it must create the correct structures to correspond to the manner in which individual words combine with each other. For example, in the case of ambiguous sentences (sentences with multiple syntactic "parses"), semantic analysis is responsible for the removal of ambiguity.

A sublanguage is a subset of natural language. The term "sublanguage" is often used to represent the language of texts in various fields, such as medicine or car repair manuals. In many ways, the processing of sublanguages has been more successful than the processing of natural language as a whole. For example, it is only within the domain of sublanguages that automatic translation has been made practical.

Information extraction systems transform information from a collection of relevant texts into a form that is more readily digested and analyzed [33]. The goal of information extraction systems is to build systems that find and link relevant information while ignoring extraneous and irrelevant information. Information extraction normally employs natural language processing techniques.

2.4.1. Syntax

To determine the syntactic structure of a sentence requires two things: the grammar, which is a formal specification of all structures of the language, and the parsing process, which is the method of analyzing a sentence to determine its structure [3][97]. A parsing algorithm can be described as a procedure that searches through various ways of combining grammatical rules to find a combination that generates a tree that could be the structure of an input sentence [3].

The two basic grammatical formalisms used in natural language parsing are context-free grammars and recursive transition networks. A variety of parsing algorithms can be used for each.

Context-free grammars are a very important class of grammars for natural language processing. The formalism of context-free grammars is powerful enough to describe most of the structure of natural language, but it is restricted enough so that efficient parsers can be built to analyze sentences [3].

To parse a sentence using a context-free grammar, it is necessary to find a way in which that sentence could have been generated from the start symbol. There are two ways this could occur. The first is "top down" parsing, which consists of beginning with the start symbols and applying the grammar rules forward until the sentence is completely parsed. The second is "bottom up" parsing, which consists of beginning with the symbol to be parsed, and applying the grammar rules backward until a parse tree with the start node as the root node has been completed. Parsing can be thought of as a special case of a search problem, with possible parses of various portions of a sentence serving as states to be searched. A parsing algorithm that seeks to provide all possible parses of an ambiguous sentence, instead of a single parse, must employ backtracking.

It is possible to design a parser that is a pure search algorithm. However, this method is very inefficient, with an algorithm that is exponential: $\mathbf{C^n}$, where **n** is the length of the sentence, and **C** (**C>1**) is a constant that depends on the particular algorithm chosen [3].

A more efficient algorithm is a chart parser, which has a worst case complexity of $\mathbf{K*n^3}$ ($\mathbf{O(n^3)}$), where **n** is the length of the sentence, and **K** is a constant dependent on the parsing

algorithm [3]. A chart parser avoids extensive backup during backtracking by storing intermediate constituents so that they can be reused along alternative parsing paths (instead of having to be regenerated).

A formalism that can be used in natural language parsing instead of a context-free grammar is a recursive transition network. A recursive transition network is like a simple transition network (or finite state machine) except that it allows arc labels to refer to sub-networks as well as word categories. A recursive transition network is equivalent in power to a context-free grammar [113] [3].

With recursive transition network parsing algorithms, the pursuance of a parse corresponds to transitions over network arcs. Both top down and bottom up parsing methods can be used, as can a chart-based parsing method (the chart is called a well-formed substring table, or WFST). These are modified from how they were presented during the discussion on context-free grammar parsing to a method appropriate for recursive transition network parsing. A recursive transition network parser using a WFST has the same complexity as the context-free grammar chart parser: $\mathbf{K*n^3}$, where **n** is the length of the sentence, and **K** is a constant dependent on the parsing algorithm.

One of the primary difficulties involving understanding of sentences in natural language is that of ambiguity. As already discussed, the same sentence can have multiple parses. Usually, this is due to word sense ambiguity. Word sense ambiguity arises since the same word can have different meanings. Consider the case of the word "fly." The word "fly" can be a verb, relating to how birds transport themselves using wings ("The birds fly away."), or it can be a noun referring to a small insect ("A fly has eyes with many facets."). Traditionally, the semantic phase of a natural language understanding system has primarily been relied on to remove ambiguity by placing the sentence in context. If one is discussing birds, then the first usage of "fly" is probably appropriate. If one is discussing the characteristics of insects, then the second usage of "fly" is appropriate.

Some removal of ambiguity occurs by the use of selectional restrictions. Selectional restrictions are specifications of the legal combinations of word senses that can occur. Selectional restrictions are employed to eliminate unlikely parses.

Another method used for ambiguity resolution is the method of statistical semantic preferences. In this method of ambiguity resolution, statistics are collected from the corpora in question on the frequency of semantic roles occurring between senses, so that the most common semantic combinations can be preferred when interpreting a sentence.

It is possible for both methods of ambiguity resolution to be employed in the same system. It should be noted that some ambiguity resolution in the form of selectional restrictions or various heuristics used to rate the parses is included in some natural language parsers. Other disambiguation will occur in the semantic phase of a natural language understanding system.

2.4.1.1. A Survey of Natural Language Parsers

The choice of a natural language parser is integral to a natural language understanding approach. The speed of a parser greatly affects the overall speed of natural language processing; also, if the parser provides good selectional restrictions, then in some cases the semantic phase need not be as complex. Three natural language parsers which typify research in this area are described below.

2.4.1.1.1. Winograd's Augmented Transition Network

Winograd [113] provided a definition of an augmented transition network to be used to implement a parser. The grammar covered many of the major structures of English, but was not complete. Many constructs such as conjunction, ellipsis, noun modifier, and prepositional phrase attachment were not included. However, the grammar did provide a good basis for the implementation of an augmented transition network parser.

2.4.1.1.2. Sleator and Temperley Natural Language Parser

The Sleator and Temperley parser is based on a formal grammatical system called a link grammar [101]. The expressive power of link grammars is equivalent to that of context-free grammars. A sequence of words is in the language of the link grammar if there is a way to draw links between words in such a way that: 1) The local requirements of each word is satisfied, 2) The links do not cross, and 3) The words form a connected graph. The linking requirements of each word is stored in a dictionary. A word such as "cat," for example, can have a backward connector allowing it to link to a determiner (such as "the" or "a"), or a backward connector that allows it to serve as an object, or a forward connector that allows it to serve as a subject. Plugging a pair of connectors together corresponds to drawing a link between that pair of words and corresponds to a partial parse of a sentence.

Coordinating conjunctions such as "and" pose a problem for link grammar parsers, since they result in links that cross. Sleator and Temperley give the following example: "Danny and Davy wrote and debugged a program." In this sentence there should be links from "Danny" to "wrote" and "debugged," and links from "Davy" to "wrote" and "debugged." The link from "Danny" to "wrote" would cross the link from "Davy" to "debugged." Sleator and Temperley have a proprietary scheme that allows the use of most conjunctions of this type.

The Sleator and Temperley parser handles: noun-verb agreement, questions, imperatives, complex and irregular verbs, different types of nouns, past-participles, present-participles, commas, various adjective types, prepositions, adverbs, relative clauses, and possessives, among others. The Sleator and Temperley parsing algorithm is $\mathbf{O(n^3)}$, where **n** is the number of words in the sentence. It can parse typical newspaper sentences in a few seconds.

The Sleator and Temperley parser uses various heuristics to weight and sort parses in the case of ambiguous sentences. The linkage (parse) considered most likely by the weighting scheme is presented as the first parse. The following heuristics, which are called linkage parse factors, are employed:

1) the sum of the lengths of the links. This is computed as the distance between the right and left connecting words.

2) the sum of the costs of the chosen disjuncts. A disjunct corresponds to one particular way of satisfying the requirements of a word. A disjunct consists of two ordered lists of connector names: the left list and the right list. The left list contains connectors that connect to the left of the current word. The right list contains connectors that connect to the right of the current word. To satisfy a disjunct, all of its connectors must be satisfied by appropriate links.

3) the imbalance, or unevenness, of the *and* lists. The cost is calculated using the number of *and* elements, and the *and* element size. It is calculated as the difference between the biggest and the smallest *and* element size in the *and* elements.

In a test run using the 1991 version of their parser on two ten-line columns from the Op-Ed page of the New York Times, the preferred linkage (the first linkage) was correct 5 out of 10 times for one column, and the preferred linkage was correct 6 out of 10 times for the other column. Their 1995 version handles 80% of Wall Street journal sentences [6].

2.4.1.1.3. Bickerton and Bralich Parser

Bickerton and Bralich [6][13] produced a natural language parser that they claimed was the fastest and most advanced parser available through 1996 [13]. They made their parser available over the World Wide Web, and issued a challenge to other parsing researchers and developers to compare their parsing tools to the Bickerton and Bralich parser.

Evaluators included Paul Deane, and Sleator and Temperley [6]. They found several problems with the Bickerton and Bralich parser. Sleator and Temperley concluded that their parser was superior to the Bickerton and Bralich parser.

2.4.2. Semantics

Producing the syntactic parse of a sentence is only the first step toward the understanding of the sentence [97]. The meaning of the sentence must still be determined. The determination of meaning is a mapping process: the sentence (and its constituent words) are mapped into some knowledge-representation form, such as a semantic net.

The first step in semantic processing is word sense disambiguation, also called lexical disambiguation. This is the process of determining the correct meaning of an individual word. Word sense ambiguity arises since the same word can have different meanings. Consider the case of the word "fly." The word "fly" can be a verb, relating to how birds transport themselves using wings ("The birds fly away."), or it can be a noun referring to a small insect ("A fly has eyes with many facets."). The semantic phase of a natural language understanding system removes word sense ambiguity by placing a sentence in context.

There are several approaches to the semantic processing of a sentence [97]:

1) semantic grammars, which combine syntactic, semantic, and pragmatic knowledge into a single set of rules in the form of a grammar. There is usually a semantic action associated with each grammar rule.

2) case grammars, in which the structure built by the parser contains some semantic information. Case grammar rules provide both syntactic and semantic structure.

3) conceptual parsing, in which syntactic and semantic knowledge are combined into a single interpretation system that is driven by the semantic knowledge. Semantic knowledge sets up expectations for particular parses of a sentence.

4) compositional semantic interpretation, in which semantic processing is applied to the parse of a sentence. In this case, syntactic parsing and semantic understanding are treated as separate steps.

One approach to disambiguation is to use spreading activation methods in semantic networks. In this case word senses that are related activate each other [3].

In addition to sentence level semantic processing, the understanding of larger linguistic units, such as paragraphs and larger texts, requires discourse analysis. There can be many semantic relationships between sentences. Pronoun resolution is a element of discourse analysis. For example, "Joan saddled the horse. John rode it." The pronoun "it" in the second sentence refers to "horse" in the first. Other types of semantic relationships between pairs of sentences include parts of entities, in which the second sentence is understood to refer to a noun in the first sentence; elements of sets, in which the second sentence is understood to refer to one or more items of a set listed in the first sentence; causal chains, in which the first sentence is understood to be the reason why the action in the second sentence took place; and several others. Similar semantic relationships appear as well in sequences of sentences greater than a pair.

Rich and Knight [97] say that there are two important parts in the process of using knowledge to provide natural language understanding (semantic analysis): first, focus on the relevant parts of the knowledge base (this focusing is important if the knowledge-base is large) and second, use that knowledge to resolve ambiguities and assist in discourse analysis.

2.4.3. Sublanguages

In Chapter IV, it is shown that comments in computer software can considered to be a sublanguage of the larger natural language. This helps significantly in reducing the complexities of the understanding process.

A sublanguage is a subset of a natural language. It is not known how many sublanguages exist in a given language [55][65]. Sublanguages emerge gradually through the use of a language in various fields by specialists in those fields. Some such sublanguages are the "language of biophysics," the "language of car repair manuals," and the "language of naval telegraphic transmissions." In many ways, the processing of sublanguages has been more successful than the processing of natural language as a whole. For example, it is within the domain of sublanguages that automatic translation has been made practical. Lehrberger states that the practicality of

automated translation of certain sublanguage texts depends on the simplified structure of the sublanguage compared with the structure of the language as a whole [72].

Sublanguage analyses most often have dealt with texts limited to specific fields, such as technical or scientific fields. However, the term sublanguage has also been used to refer to "the sentences of a language closed under some or all of the operations in the language" [71]. This definition restricts the format of the language but does not restrict the semantic domain of the language. Lehrberger [71] refers to an information sublanguage as a sublanguage whose semantic domain is no more restricted than that of natural language itself. He refers to a sublanguage with a restricted semantic domain as a subject-matter sublanguage.

Lehrberger [71] says that "grammaticality in a subject-matter sublanguage is determined by whatever officially prescribed or implicit norms of usage exist among the specialists in the subject-matter field." Some grammatical restrictions in sublanguages may involve the use of the imperative, the common sentence length, uniformity of tense, modality, the use of the interrogative, the deletion of articles (telegraphic), the deletion of the object noun phrase, the use of proper nouns, and many others [64][71].

In some cases the norms of usage may conflict with those of standard language. Sublanguage structures that do not conform to standard language are referred to as "deviant." However, normally deviant sublanguage structures can be paraphrased in the standard language, and could often be replaced by standard language structures. One fairly common example of deviant sublanguage structures is the telegraphic style that is used in weather bulletins, or Navy telegraphic messages [71][52][65].

Sublanguage texts usually contain some material that does not belong to the sublanguage proper. This may occur as whole sentences interspersed among those of the sublanguage proper, or as matrix expressions. Lehrberger [71] gives the following example drawn from a geometry text:

> "Pythagoras proved the theorem that now bears his name in the sixth century B.C. He was able to show that the area of the square on the hypotenuse of a right triangle equals the sum of the areas of the squares on the other two sides."

The first sentence is a sentence that is not properly a sentence about geometry, but of history. Also, the phrase "He was able to show that," which is called a matrix expression, is not part of the sublanguage of geometry. Lehrberger concludes that this is a general phenomenon in real texts from restricted sublanguages, and that a text typically consists of a mixture of discourse from within a particular domain, and metadiscourse about it.

In summary, the factors which can help to categorize a sublanguage are [71]: 1) limited subject matter, 2) lexical, syntactic, and semantic restrictions, 3) "deviant" rules of grammar, 4) high frequency of certain constructions and low frequency of certain constructions, 5) text structures, and 6) the use of special symbols. It will be shown that comments in computer software meet all of these criteria.

The fact that a sublanguage consists of limited subject matter means that a semantic network employed in the analysis of the sublanguage can be simpler than one employed to analyze general natural language. It also means that a word-based natural language parser used in the sublanguage analysis would have a smaller dictionary, and thus suffer much less from parse ambiguity (multiple parses). The lexical and syntactic restrictions of a sublanguage means that a parser used in the sublanguage analysis need not handle all the possible constructs of the natural language as a whole, but can support a simpler subset of the natural language. Since sublanguages sometimes employ deviant rules of grammar and certain special symbols, the natural language parser must be modified to handle these additional rules and symbols.

2.4.4. Information Extraction Systems

The purpose of information extraction (IE) systems is to transform information into a form that is more readily digested and analyzed. Cowie and Lehnert [33] say, "the goal of IE research is to build systems that find and link relevant information while ignoring extraneous and irrelevant information." Information extraction systems normally employ natural language processing techniques on text data in electronic form. This has several similarities to the problem of program understanding, particularly when the processing of comments and identifiers from computer

software is considered as the primary source of information. The measurement techniques used in the area of information extraction can be adapted to measure comment and identifier understanding, and provide a framework from which evaluation criteria can be derived.

There are several important aspects of information extraction systems from the point of view of natural language researchers [33]. First, the information extraction tasks are well-defined. Second, information extraction tasks use real-world texts. Finally, information extraction performance can be compared to human performance on the same task.

Early information extraction systems included those of DeJong [36][33] and Zarri [115][33]. DeJong's FRUMP system analyzed news stories using simple scripts. FRUMP attempted to match each new story with a relevant script on the basis of keywords and conceptual sentence analysis. Zarri's system used texts that described the activities of various French historical figures. The system sought to extract information about the relationships and meetings between these people.

Some commercial systems included ATRANS [79][33], which was designed to handle money transfer telexes in international banking. It used sentence analysis techniques similar to those in FRUMP. ATRANS demonstrated that relatively simple language processing techniques are adequate for IE applications which are narrow in scope and utility. Another commercial system was JASPER [4][33]. JASPER was designed to extract information from reports on corporate earnings. It achieved its natural language parsing capabilities through template-driven techniques for language analysis operating on small sentence fragments.

Several Information Extraction systems were evaluated in the Message Understanding (MUC) conferences [34][5]. The outputs of the Information Extraction systems which were evaluated were a set of filled-in templates. The evaluation employed four primary measures of performance [34]: recall, precision, overgeneration, and fallout. Recall is a measure of the completeness of the template fill. Precision is a measure of the accuracy of the template fill. Overgeneration is a measure of spurious generation. Fallout is a measure of the false alarm rate for slots that could be filled only from a finite set of slot fillers. In MUC3 these were defined as follows [34]:

The **number of possible slot fills** was the number of slot fills in the key plus the number of optional slot fills in the key that were matched by the response.

The **number of actual slot fills** was the number of slot fillers in the response.

recall = (Correct + Partial * 0.5) / possible

precision = (Correct + Partial * 0.5) / actual

overgeneration = Spurious / actual

fallout = (Incorrect + Partial) / possible incorrect

where possible incorrect is the number of possible incorrect numbers that could be given in the response. Fallout is a measure of the false alarm rate, and the number of false alarms could only be measured for slots where there were a finite number of possible responses (the number of possible incorrect responses was known).

The measures for recall, precision, and overgeneration can be applied to the approach described in Chapter III. However, the measures must be modified to operate on a non-template based system. The fallout measure is applicable only to a template-based system, where the maximum number of responses is known, so it is not employed in the approach described in this dissertation.

2.5. Semantic Networks

A semantic network is a structure for representing knowledge as a pattern of interconnected nodes and arcs [103][104]. Semantic networks are commonly used for the knowledge-base in natural language understanding systems, and are used in the approach discussed in Chapter III.

Semantic networks were originally developed for representing semantic information, either static information about concepts, or information about the meaning of natural language sentences [10]. Dozens of different versions of semantic networks have been designed and implemented. Semantic networks share two general features [10]: 1) The nodes represent physical or non-physical entities in the world, classes or kinds of entities, relationships, or concepts. 2) The

links, usually directed links, represent explicit relationships between nodes, where the type of relationship is specified by the type of link.

Conceptual graphs form a knowledge representation language based on linguistics, psychology, and philosophy [104]. Within a conceptual graph, the roles that a concept plays with respect to other concepts are specified. A conceptual graph only has meaning within the context of a semantic network.

Barnden [10] characterizes semantic networks into two rough categories: restricted or general. In the restricted type network, action relationships between nodes such as love or eat are represented by links. However, the most commonly used types of links in a restricted type network are IS-A links and INST links. The IS-A type link says that one node is a sub-kind of the other node, whereas INST says that one node is an example of a kind that is represented by the other node. The purpose of a restricted semantic network is to organize concepts, kinds, or classes into a hierarchical taxonomy. In a hierarchical taxonomy, information represented at a node (by links to other nodes) can be inherited by that node's subnodes. Also, information represented at a node (by links to other nodes) can contradict information inherited from that node's ancestors.

In a general semantic network, links have a close relationship to deep-case relationships in natural language semantic theories. Examples of such link labels are AGENT link and OBJECT link labels. The AGENT link connects the agent of an action to the action performed. The OBJECT link connects the action to the entity which is acted upon. Action relationships between nodes such as love or eat are represented by nodes rather than by links. This makes the semantic network structure more general, since defining a new relationship does not require defining a new type of link. The more elaborate general semantic networks provide inferencing schemes that are equivalent to first order logic.

Spreading activation is one common method of inferencing within a semantic net. With spreading activation, when a node becomes active, the nodes adjacent to that node, and directly connected to that node, can also become active. Activation is usually a yes-no matter as opposed

to the graded activation that is typically used in neural networks. However, graded activation is also used in semantic networks [10].

The main motivation for spreading activation is intersection search. In a semantic network intended for natural language processing, an intersection, or path between word senses aids in the disambiguation of words.

A semantic network differs from a neural network in several ways [10]. In neural networks, spreading activation is the only mechanism for short term computation. This is not normally the case in general semantic networks. The type of activation spread is normally more complex in a general semantic network than in a neural network. The topology of a semantic network can be changed by processing mechanisms in arbitrarily extensive and complex ways in the short term. This is not true of a neural network [10]. Neural networks do not have type labels on links, so that neural networks normally do not treat different input links differently. Neural networks do not directly support conceptually asymmetric relationships. For these reasons, semantic networks are normally more appropriate for natural language understanding than are neural networks.

2.5.1. Conceptual Graphs

Conceptual graphs are often used to form semantic networks, and are also employed in semantic networks that provide natural language generation capability. Thus, the use of conceptual graphs provides a somewhat standardized form for a semantic network, which is easily employed in natural language generation. The approach described in Chapter III employs conceptual graphs in a semantic network which is also used in natural language generation.

Conceptual graphs form a knowledge representation language based on linguistics, psychology, and philosophy [104]. A conceptual graph only has meaning within the context of a semantic network. A conceptual graph is a finite, connected, bipartite graph [104]. They are *connected* since two unconnected parts would simply be two conceptual graphs. They are *bipartite* because they consist of two different kinds of nodes, *concepts* and *conceptual relations*, and every arc links a node of one kind to a node of another kind.

A definition of *concept* [109] from the research area of Formal Concept Analysis defines a concept as a "unit of thought" consisting of two parts: 1) its extension, which consists of all objects belonging to the concept, and 2) its intension, which consists of all attributes belonging to the concept. Within the area of conceptual structures, a *concept* is an internal interpretation of a perception of an entity. For example, a cat is a concrete entity. The cat could be perceived through some form of sensory perception (such as eyesight). The perception of the cat is translated into an internal concept (in the case of a person seeing the cat, the concept would be a mental concept).

A *conceptual relation* specifies the role that a concept plays with respect to another concept. An example of a concept might be [104]:

[CAT]-->(STAT)-->[SIT]-->(LOC)-->[MAT]

In this example, the words in square brackets represent concepts, whereas the arrows together with the words in parentheses represent conceptual relations (they might also be represented with boxes and circles, respectively). This conceptual graph states literally that a cat is in the state of sitting, and the sitting takes place at a location mat. In other words, a cat is sitting on a mat. A specific cat or mat is not specified in this particular conceptual graph.

Conceptual graphs as defined by Sowa [104] obeyed four canonical formation rules for deriving a conceptual graph *w* from conceptual graphs *u* and *v*: 1) copy--*w* is an exact copy of *u*, 2) restrict--for any concept *c* in *u*, type(*c*) may be replaced by a subtype, 3) join--if a concept *c* in *u* is identical to a concept *d* in *v*, then *w* is the graph obtained by deleting *d* and linking to *c* all arcs of conceptual relations that had been linked to *d*, and 4) simplify--if conceptual relations *r* and *s* in the graph *u* are duplicates, then one of them may be deleted from *u* along with all of its arcs.

2.5.1.1. Natural Language Generation using Conceptual Graphs

The approach described in Chapter III employs natural language generation using conceptual graphs to provide a reusability report for code components. The current state of research in this area can be gleaned from looking at the following system.

Velardi, Pazienza, and De' Giovanetti developed a system at IBM for analyzing and generating Italian text [108]. In their system, they used detailed semantic knowledge on word-sense patterns to relate the linguistic structure of a sentence to a conceptual graph representation. They used a conceptual lexicon of word-sense descriptions, where a description included the surface semantic pattern of the word-sense. Surface semantic patterns expressed both semantic constraints and word-usage information.

To generate a sentence out of a purely semantic conceptual graph representation, their system made several decisions: 1) It chose active or passive form, 2) It chose synthetic or direct replacement ("A meeting between the delegates" versus "The delegates participate in a meeting"), 3) It chose an emphasis ("the items of a statute" versus "a statute with items"), 4) It selected an ordering (choice of whether an adjective comes before or after a noun, which is appropriate for Italian, but not for English), and 5) It made a choice of synonyms. Their system generated a coordinate sentence construction ("John and Joe") whenever a concept was related by the same conceptual relation to different concepts.

In order to restore morphological information and attach determiners to nouns, they applied the following rules: 1) The number of a noun was determined from that noun's concept referent, 2) The number of a verb was determined from that verb's subject concept and the tense from the time/mode conceptual relations, 3) The gender (appropriate for Italian) and number of the adjective was determined from the noun modified by the adjective, and 4) A definite article was used if the referent of the concept associated with the noun being modified referred to a specific item, an indefinite article was used if the referent of the concept associated with the noun being modified was unspecific. The stem and the morphological data derived by these rules were the

input condition for morphological synthesis. Words were decomposed into prefix, stem, and suffix, and the appropriate word ending was derived from a table of word ending models.

The approach described in Chapter III uses some techniques similar to those described above, such as article determination and choice of active or passive form, in the natural language generation of a particular report, although it provides a simplified version of natural language generation.

2.6. Reusability Metrics

One critical aspect of the software reuse process is that of determining the extent to which a software component is reusable through the use of metrics which predict reusability. Reusability is typically predicted by the use of a software quality metrics hierarchy [80][92][98]. The quality factors in the software quality metrics hierarchy are normally predicted by lower level software metrics.

As an example, the Boeing/RADC Software Interoperability and Reusability Guidebook for Software Quality Measurement [92] lists reusability as a software quality factor, and application independence, document accessibility, functional scope, generality, independence, modularity, self-descriptiveness, simplicity, and system clarity as quality sub-factors. Several of the sub-factors could be considered as contributing to the software component's understandability and modifiability. However, the Boeing/RADC software quality hierarchy is typical of a *general* software quality metrics hierarchy that was not primarily concerned with reusability. Reusability was one quality factor among many. More recently, some software quality factors hierarchies have been designed that are intended to measure reusability alone, and these will be described below.

2.6.1. A Survey of Software Reusability Metrics Literature

The following software quality factors hierarchies describe design goals, factors, and strategies that are concerned solely with the measurement of reusability. Traits that should be present in reusable software and how they should be measured are discussed.

2.6.1.1. Global Protection Against Limited Strikes (GPALS) Software Reuse Strategy

The philosophy behind the GPALS software reuse strategy [54] is embodied in the following quote:

> "reusability metrics are the necessary properties of reusable software only, i. e. , general quality and correctness are assumed as a minimum and as a given. Potential reusability of the software may be considered as an aspect of quality, but quality is not a specific aspect of reusability; it is necessary, but not sufficient."

The GPALS software reuse strategy [54] listed the following specific design goals that constitute the overall goal of "reusability":

1) Abstraction--the paradigm of the component's function presented to the external interface.

2) Adaptability/Flexibility--the ease with which software can be modified to meet new requirements.

3) Cohesiveness--the degree to which a component's structure is unified in support of its function.

4) Completeness--the degree to which the component implements all required capabilities.

5) Coupling/Interconnectivity--the degree of connectivity between different components (should be minimized).

6) Expandability/Augmentability--the ability to support expansion of data and storage requirements.

7) Generality--the breadth of applicability of the component.

8) Information Hiding--the extent to which internal data objects are concealed from the external interface.

9) Modularity--the way the component is decomposed into sub-components.

10) Performance/Efficiency--empirical execution data, including history of performance.

11) Portability--operating environment independence.

12) Robustness/Fault Tolerance--ability to produce correct results despite input errors.

13) Size--as measured by source statements, lines of code, function points, or other methods.

14) Understandability/Clarity/Self-Descriptiveness--ease of comprehending the meaning of the software.

Expandability and Adaptability are related traits. Both are concerned with modifying software for use in a new environment. Completeness is also related to Expandability and Adaptability--it measures the mapping of software from an old environment to a new environment--a software component that is fully complete would exactly match the requirements of a new system. An incomplete software component would thus have to be Expandable and/or Adaptable. Generality is related to completeness in that a generalized component would have a high measure for completeness in many other systems. Understandability is related to Generality and Completeness in that it is difficult to determine the breadth of application of a software component if it cannot be understood. Information hiding, Coupling, Cohesiveness, Modularity, and Portability are related to how hard the component is to extract from its original system. Robustness and Size are related to the reliability of the component. Good reliability is always desirable for reusable code.

The GPALS software reuse strategy did not specifically address how these quality factors were to be measured, which was the focus of the next study.

2.6.1.2. Asdjodi-Mohadjer

Asdjodi-Mohadjer suggested a process and an approach for evaluating the reusability of software components[7]. The reuse measurement process included the following steps: 1) Establish reuse requirements/factors, 2) Define reuse metrics, 3) Implement the reuse quality metrics, 4) Analyze the results, and 5) Validate the results.

Asdjodi-Mohadjer noted that the reusability of software can be measured from different points of view. Most commonly, software reusability has been measured on the basis of good programming practices, or from a history of error-occurrence. However, from a user's point of view the quality is defined as customer satisfaction.

When considering the reusability of a particular software component, Asdjodi-Mohadjer says that, in general, "the reusability of a software system or component depends on its modification level when the original requirements change." The magnitude of a reusable component can range from a simple function (reuse-in-the-small) to a large system (reuse-in-the-large).

Asdjodi-Mohadjer lists the following reuse factors for software components or systems:

1) Modularization/Layering/Modeling

2) Encapsulation (separation and completeness of related functionalities and entities)

3) Abstraction (representation of ideas against specific instances, implementation independent features, flexibility toward change of requirements)

4) Parameterization (overlaying of names, inheritance of generic high level classes, object-based interfaces)

5) Environment independence/installability (e. g. operating system, hardware)

6) Tool independence/clean interface

7) Use of reusable/generic tools and software

8) Clear module interface/standardization of interfaces

9) Reusable/generic functionalities

10) Independence of execution/parallel execution

11) Virtual Machine and Networking (loosely coupled)

12) Quality of the design and development (good programming practices)

13) Availability of the parts and documentation

14) Popularity/usage of the component

15) Reliability of the design, code, and data

16) Reliability/Maturity of the development process

17) Document generation approach

18) System composition approach

Asdjodi-Mohadjer looks at high level reusability criteria, and low level reusability criteria. She provides several criteria for determining the reusability of domain engineering, and the reusability of a system design in the Command and Control (C^2) domain. She provides several other criteria for evaluating the reusability of software. Her software reusability criteria include a criterion that asks if the software is commented properly, and another criterion that asks if the complexity of the software is acceptable. Yet other criteria ask if the encapsulation is appropriate and if the programming style is acceptable.

The reuse quality factors defined by Asdjodi-Mohadjer are very similar to those employed in the GPALS reuse strategy. Modularization maps to GPALS Modularity. Encapsulation is similar to GPALS Cohesiveness, Completeness, and Information hiding. Reliability is similar to GPALS Robustness. There are several other similarities, although Asdjodi-Mohadjer in her list of reuse quality factors is taking a software system-wide view of reusability. Thus her list of reuse quality factors also includes such items as Document generation approach and System composition approach. However, she also provided an example of a reusability quality determination in a particular domain (the C^2 domain), where the criteria were targeted at the software component level, and where the criteria were derived from the higher level reusability quality factors. She suggests various low level metrics that can be used to measure these reusability criteria.

2.7. Software Metrics for Object-Oriented Software

Since reusability assessment requires low level metrics to predict higher level software quality factors, it is important to have not only the correct metrics but the proper relationships among metrics. Earlier defined low level metrics have traditionally been metrics intended for use for software developed in the functional decomposition paradigm, such as McCabe's cyclomatic complexity metric [81]. However, software developed using the object-oriented paradigm deals with significantly different concepts. Two approaches have been taken for measuring object-oriented software. The first approach has concentrated on attempting to predict maintainability and reliability by using traditional complexity metrics and showing how those measures relate to object-oriented software. The second approach has been to develop new object-oriented specific metrics. Both approaches will be discussed below.

2.7.1. Traditional Complexity Metrics Applied to Object-Oriented Code

Tegarden and Sheetz [107] have identified problems with the application of traditional software metrics such as lines of code, McCabe's cyclomatic complexity, and Halstead's software science to object-oriented programs.

For lines of code, the questions that must be resolved are: 1) Should the lines of code of an inherited method be counted as part of every object in the hierarchy that inherits it, or just the objects that use the method? Or should lines of code be counted only once at the level of definition? 2) Should the lines of code of class and instance methods be counted differently? 3) When polymorphism is used, should the lines of code be summed over all definitions of the method, or should each method be viewed independently of the other methods? 4) How much of the class libraries used should be counted?

When considering the use of Halstead's software science, Tegarden and Sheetz assume that methods correspond to operators, and that variables correspond to operands. The problems that arise are: 1) Should an inherited class/instance method or variable count in each object that

inherits it, the object that uses it, or only in the object that defines it? 2) Is each polymorphic definition of a method a separate operator, or an increment to the count of that operator at the highest level in the object hierarchy?

When considering the use of McCabe's cyclomatic complexity, they determined that a system level cyclomatic complexity cannot be calculated by summing the cyclomatic complexity of all the methods in subordinate objects, because not all contributions to the complexity of a method can be derived from subordinates. Some methods are inherited from superclasses, some methods are used from other parts of the object hierarchy, and some methods are used from the object. To sum the constituent values into a system level metric, consistent with the calculation mechanism employed in systems designed using the functional-decomposition paradigm, requires the generation of the metric for all objects in the inheritance hierarchy. They mention that some decisions normally made locally in a structured module would, in an object-oriented system, be passed through a message to other objects. Thus, they expect that methods in object-oriented systems would tend to have low values for cyclomatic complexity.

Tegarden and Sheetz also calculated the traditional metrics for four systems written in C++: 1) a system that included neither polymorphism or inheritance, 2) a system that included only polymorphism, 3) a system that included only inheritance, and 4) a system that included both polymorphism and inheritance. They determined that the use of inheritance or polymorphism tends to decrease the complexity of an object-oriented system. They concluded that traditional metrics can be used on object-oriented systems, although the magnitude of these metrics when used for comparison to systems that were designed in the functional-decomposition paradigm is suspect.

2.7.2. Object-Oriented Design Metrics

In several cases, object-oriented metrics are more appropriate when applied to the design of object-oriented software than its implementation. For example, the height (or depth) of the inheritance hierarchy can be used to determine the amount of coupling, and in determining

maintainability. This metric can be derived at design time, as can most of the object-oriented metrics that have currently been defined.

The study of object-oriented metrics and object-oriented design techniques leads to the conclusion that object-oriented software itself maps more closely to its design than software developed using the functional-decomposition paradigm. This is advantageous since it makes an evaluation of the current design practicable, allowing improvements before that design has been converted into code. This also means that meaningful design metrics values can be calculated by an examination of the source code, which supports an assessment of modifiability for reuse. Good values for design metrics imply that the modification will be easier.

2.7.3. A Survey of Object-Oriented Design Metrics

Several different object-oriented design metrics suites have been proposed and are discussed in this section. A complete, validated set of object-oriented design metrics does not exist.

2.7.3.1. Chidamber and Kemerer

Chidamber and Kemerer [26][27] proposed six object-oriented design metrics:

1) DIT--depth of inheritance tree. This metric measures the distance from the base class to the bottommost children in an inheritance tree.

The lower a class is in the inheritance tree, the more superclass properties it may inherit, and thus the complexity is larger. High values of DIT tend to result in high values of complexity and low values of maintainability.

2) NOC--number of children. This metric measures the number of classes that inherit the current class.

The more children a class has, the more classes it may affect if modified (because of inheritance). This tends to result in lower maintainability.

3) RFC--response for a class. The response set for a class is a measure 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 methods. The cohesion of a class is characterized by how closely the local methods are related to the local instance variables.

Low cohesion increases complexity. Thus, the larger the lack of cohesion, the lower the maintainability.

5) WMC--weighted methods per class. The WMC is the sum of the complexities of all local methods. If all method complexities are considered to be unity, then WMC = the number of methods inthe class.

The more methods, and the more complex the methods, the more complex the class. Thus the larger the WMC, the lower the maintainability. Objects with large numbers of methods are likely to be more application specific, which limits the possibility of reuse.

6) CBO--coupling between objects. The CBO for a class is a count of the non-inheritance related couples with other classes. Two classes are coupled when the methods of one class use methods or instance variables of another class.

The calculation of some of these metrics is straightforward--such as the calculation of the DIT and NOC metrics. The calculation of other metrics is less straightforward. Indeed, some of the metrics have more than one definition. For example, the lack of cohesion of methods metric (LCOM) has been defined two different ways by Chidamber and Kemerer [27].

2.7.3.2. Li and Henry

In their study of the use of object-oriented metrics to determine maintainability, Li and Henry [74][75][76] used five of the six object-oriented metrics proposed by Chidamber and Kemerer:

depth of inheritance tree (DIT), number of children (NOC), response for a class (RFC), lack of cohesion of methods (LCOM), and weighted methods per class (WMC). They did not use the coupling between objects (CBO) metric because they considered Chidamber and Kemerer's definition of CBO as the "count of non-inheritance-related couplings" to be vague, since they felt that there is more than one form of non-inheritance-related coupling in various object-oriented languages.

Li and Henry defined LCOM as: LCOM is equal to the number of disjoint sets of local methods. No two disjoint sets intersect (by definition); any two methods in one disjoint set access at least one common local instance variable; the number of local instance variables accessed ranges from 0 to N, where N is a positive integer.

Li and Henry used the definition of the weighted method per class (WMC) metric as defined by Chidamber and Kemerer. However, they chose the measure of complexity of a method to be the McCabe's cyclomatic complexity metric. Thus WMC was calculated as the sum of the McCabe's cyclomatic complexities of each local method.

They defined some additional coupling metrics: 1) coupling through inheritance, 2) coupling through message passing, and 3) coupling through abstract data types. They used DIT and NOC to measure coupling through inheritance (in addition to their use as a measure of complexity). They defined a new metric called MPC (message passing coupling). MPC was defined as the number of messages sent out from a class. The number of messages sent out from a class indicates how dependent the implementation of the local methods is on the methods in other classes. They defined a new measure called Data Abstraction Coupling (DAC) to measure coupling through abstract data types. A variable within a particular class may have a type which is an abstract data type; that is, a variable within a particular class may be an instance of another class. DAC was defined as the number of abstract data types within a class.

Li and Henry [74][75][76] also defined a metric that was intended to measure the size of the interface of a class. The interface of a class is the set of operations, or local methods, defined in the class. The metric used to measure the incremental interface of a class is the number of

methods metric (NOM). They also defined two size metrics: SIZE1, the number of semicolons in a class, and SIZE2, the number of attributes plus the number of local methods.

A validation study was performed on two commercial products implemented in Classic-Ada [74][75] (Classic-Ada is an object-oriented version of Ada). The maintenance effort was measured by the number of lines changed per class, and was collected over three years. The results of the validation showed a strong correlation of the metrics values to the maintenance effort. They concluded that software designs can be measured quantitatively using software metrics in the object-oriented paradigm, and that software metrics collected from the design can predict maintenance effort.

2.7.3.3. Rajaraman and Lyu

Rajaraman and Lyu [94] proposed four coupling metrics, intended primarily for C++ software, but that could be extended to other object-oriented languages. These metrics were to be used in calculating reusability and maintainability. These four metrics were:

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 (CNIC)

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 ancestors. A way in C++ in which non-inheritance related coupling can occur is by the use of friend functions or classes.

3) Class Coupling (CC)

Class Coupling (CC) is the sum of the method couplings (MC) of all methods in the class, where method coupling is defined as:

$$\text{Method coupling (MC)} = \text{number of non-local references}$$
$$= gv + gf + om + iv$$

where: *gv* = number of global variable references

gf = number of global function uses

om = number of messages to other classes

iv = number of 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

The above metrics were validated in four different C++ systems versus a ranking of the classes by the developers in the order of perceived testing and maintenance. Their validation experiment showed that CC and AMC correlated better with perceived testing and maintenance difficulty than the traditional complexity metrics such as McCabe's cyclomatic complexity and Halstead's Software Science.

2.7.3.4. Lorenz and Kidd

Lorenz and Kidd [78] collected object-oriented metrics for several years over a large number of Smalltalk and C++ projects. They defined and collected metrics at the project level, and at the design level. Examples of project level metrics they collected were number of key classes, number of support classes, and average number of support classes per key class, where a key class is a class that is central to the domain being developed.

For the design level metrics, Lorenz and Kidd collected metrics designed to measure the following areas: method size, method internals, class size, class inheritance, method inheritance, class internals, and class externals. For class size, they used metrics relating to the number of instance variables and methods in a class. They used the number of public instance methods as an indicator of the number of services being provided by the class to other classes. For method size, they used the number of message sends, and the number of lines of code, among others. For method internals they used strings of message sends, and method complexity, which they

calculated using a variety of measures. Among these measures were the number of parameters, the number of nested expressions, the number of temporary variables, and etc. For class inheritance, they used the depth of the inheritance tree, among others. For method inheritance, they used the number of methods overridden by a subclass, the number of methods inherited by a subclass, and the number of methods added by a subclass. For class internals they used the use of friend functions, the average number of parameters per method, the average number of comment lines per method, and the average number of commented methods, among others.

Lorenz and Kidd provided threshold numbers for many metrics discussed in their book. Thesethreshold numbers reflected empirical data derived using these metrics over many Smalltalk and C++ projects. They mention that these thresholds are heuristics, and not absolute numbers.

2.7.3.5. Bansiya and Davis

Bansiya and Davis [9] employed object-oriented metrics in an assessment of high level quality metrics. They defined several "design properties," i.e., tangible concepts that can be directly assessed by the examination of the internal and external structure, relationships, and functionality of the design components. These design properties were: 1) abstraction--a measure of the generalization-specialization aspect of the design, 2) encapsulation--the enclosing of data within a single construct, 3) modularity--a property that relates to developing classes with a comparable number of services, 4) coupling--the interdependency of an object with other objects in the design, 5) cohesion--assesses the relatedness of methods and attributes in a class, 6) composition--reflects aggregation relationships (such as part-of, or consist-of) in an object-oriented design, 7) inheritance--a measure of the is-a relationship between classes, 8) polymorphism--the measure of services that are dynamically determined at run-time in an object, 9) messaging--a measure of public methods that are available as services to other classes, and 10) complexity--the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Bansiya and Davis specify a comprehensive set of design metrics to cover all the above design properties. These design metrics are derived metrics; that is, they combine results from simple counting metrics (such as depth of inheritance tree) using standard statistical methods. To measure abstraction, a design metric called Average Number of Ancestors (ANA) was used. Encapsulation was measured by a metric called Data Access Metric (DAM). Modularity was measured by a metric called Measure of Functional Modularity (MFM). Coupling was measured by Average Direct Coupling (ADC), and cohesion by Average Class Cohesion (ACC). Composition was measured by a metric called Measure of Aggregation (MOA). Inheritance was measured by Average Depth of Inheritance (ADI). Polymorphism was measured by a Number of Overridden Methods (ANM), and complexity by Mean Class Complexity (MCC). Most of these metrics were not further defined in the Bansiya and Davis paper; however, it was mentioned that the average depth of inheritance metric (ADI) is computed by dividing the summation of nesting levels of all classes by the number of classes.

Finally, Bansiya and Davis employed the design properties in a software quality metrics model, to calculate values for software quality factors such as Reusability and Maintainability. They performed a validation test over 18 object-oriented project designs. Based on this study, they concluded that quality assessment using an approach employing design metrics to calculate high level quality attributes is a non-intrusive and easy to apply process.

Chapter III

APPROACH TO THE AUTOMATED IDENTIFICATION OF REUSABLE COMPONENTS IN OBJECT-ORIENTED CODE

This research addresses the reuse of object-oriented legacy software by the development of an automated approach to the identification and qualification of object-oriented software components. This chapter will provide an overview of the research goals and choices that led to the final research approach.

3.1 Reuse Concept and Research Goals

The process of identifying potentially reusable components in legacy code, and of determining whether each component has sufficient quality to make reusing it worthwhile has traditionally been performed in an ad-hoc manner by software developers. One primary goal of this research is to develop and demonstrate an automated approach for the identification and qualification of reusable components. These techniques will make the identification of potentially reusable code faster, more accurate, more consistent, and more affordable. The software to be analyzed is object-oriented legacy code that has not specifically been prepared for later reuse. A large body of such code currently exists, and more is being created each day[83][84].

Figure 1 represents the processing stages that would be required in order to prepare classes from object-oriented legacy code for insertion into a reusable component repository. This includes 3 major phases: 1) an understanding, or identification, phase, which is required to understand a class and thus identify whether or not that class is useful in the domain of interest,

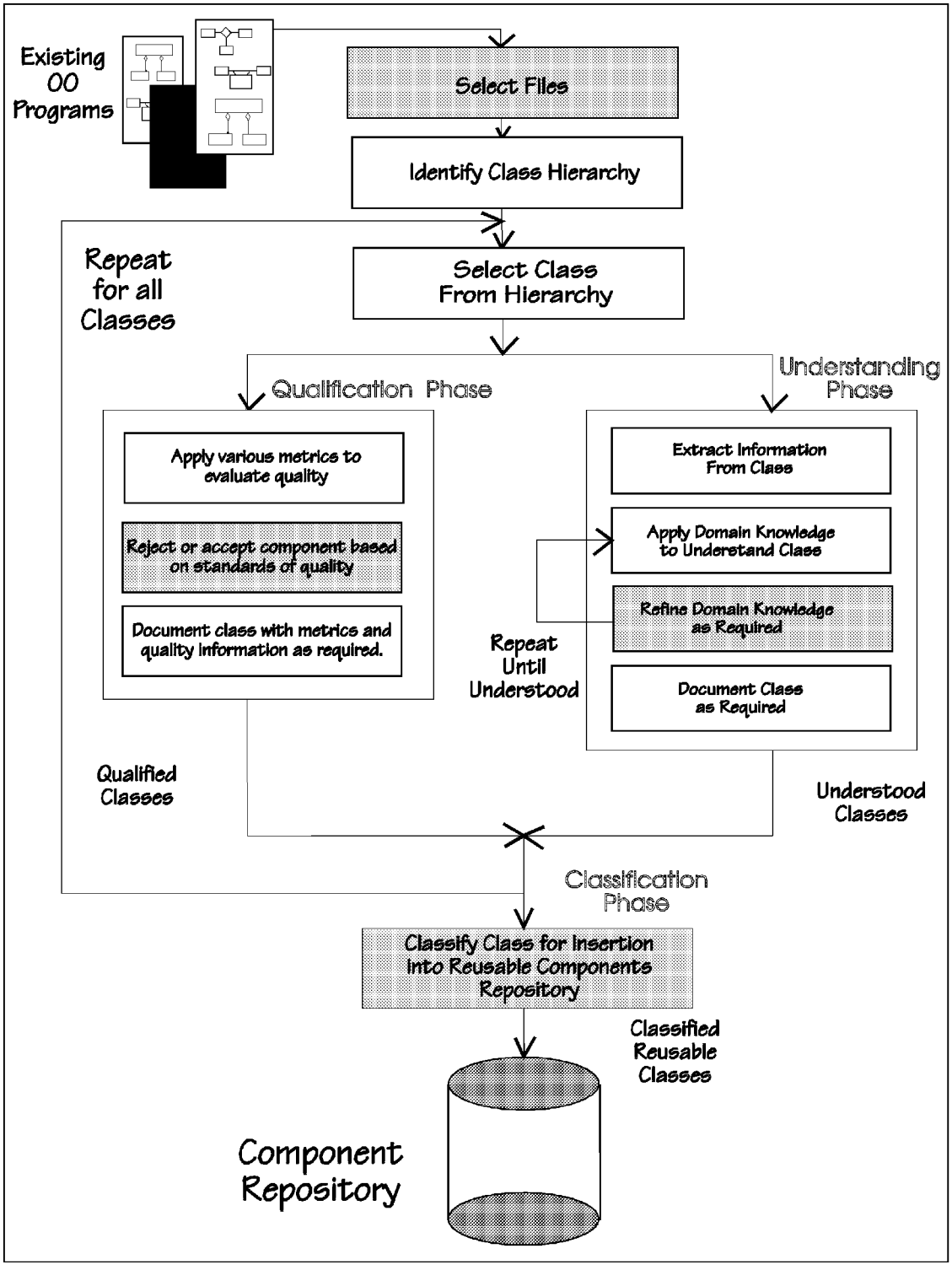


Figure 1. The Process of Identifying, Qualifying, and Classifying Reusable Components in Object-Oriented Legacy Code

2) a qualification phase, which is required to determine whether or not a class is of sufficient quality to be reused, and 3) a classification phase, which classifies the class so that it can be inserted into a code component repository and retrieved later for reuse. The understanding phases and qualification phases are shown occurring in parallel in Figure 1, since they can be implemented as independent operations.

The approach described in this dissertation automates as many of these tasks as was practical. The shaded boxes in Figure 1 are the only tasks that have not been automated. Manual tasks include updating the domain knowledge by learning (automating this process is a possible future research topic). Also, the rejection or acceptance of components based on the values of reusability quality metrics still requires a knowledgeable person. A knowledge-based assistant for this task could also be a fruitful area of further research. The automated classification of reusable components, based upon information provided by the system is also a future research topic.

To achieve the goals of this approach, four main sub-problems had to be solved: 1) automating the understanding of the functionality of a particular software component, 2) determining an appropriate set of metrics to estimate how reusable the software is, 3) automating the understanding and metrics assessment process, and 4) validating that reasonable and predictable results can be obtained using this approach. Each of these areas will be described in the following sections.

3.2. Choices Made for the Program Understanding Approach

One major goal for this research was to automate the identification and qualification process as much as possible. Thus the previously described algorithmic approach [24], which normally requires the intervention of a software engineer, was not appealing. The transformational approach suffered from the previously mentioned problems [73]: rules must be matched only to strict syntactic form, which makes the knowledge-base very large; also, it is possible to have a "transformation cycle," where part of a program is transformed back into its original form; and

finally, a transformation rule can not match to non-adjacent program components. Therefore, techniques from the knowledge-based approaches appreared to have the greatest potential for success.

Two of the three primary knowlege-based approaches, the graph-parsing and heuristic concept recognition approaches, possess several difficulties: first, these approaches require a fairly large knowledge base for the understanding of even small programs, and thus can be difficult to apply to large systems, and second, domain level knowledge is not addressed by either of these approaches.

Therefore, since the approach needed to work on large software systems, a knowledge-based, heuristic, approach to program understanding was selected, specifically, the informal tokens approach. An approach based on informal tokens taken from comments and identifiers is particularly appropriate for object-oriented software. For object-oriented software, more so than for functionally-oriented software, much understanding can be performed by simply looking at comments and identifier names [41][42][48][50]. This is true since object-oriented software is organized in classes, with everything required to implement a class at least mentioned (if not defined) in the class definition. Figure 2 represents a C++ class whose purpose is serial communications. The member functions associated with this class would largely consist of writes-to and reads-from various hardware locations. The hardware writes and reads would be extremely difficult to understand without a schematic of the underlying hardware. Obviously in this case a maintenance person, or a software engineer considering inclusion of the class and its member functions in a software reuse library, would be heavily dependent on the information received from comments and identifiers.

Informal tokens by their nature are related to the natural language of the software developer. Previous approaches using informal tokens failed to address issues important to natural language processing. They did not formalize an approach to comment analysis as part of natural language understanding. They provided no syntactical analysis; rather, they consisted of a keyword semantic analysis of comments and identifiers. The understanding approach in this research differs from earlier informal tokens approaches [16][17][18][19][20][21] in many ways, which

```
/* This class provides serial, polled I/O drivers */

class com {

    unsigned num_data_bits;
                    /* Number of data bits */

    unsigned parity; // None = 0, even = 1, odd = 2

    public:

    unsigned in_data(); // Read serial I/O

     void out_data (unsigned val);
       /* This routine provides a serial I/O write */

 }
```

Figure 2. The Necessity for Informal Tokens in Program Understanding

eliminate these shortcomings. This is done by treating comments in computer programs as a grammatical and subject-matter sublanguage [41] and using natural language processing techniques. This approach uses as a basis a study of identifiers in C++ programs that identifies certain common formats which allow the syntactic tagging of identifiers and the use of natural

language processing techniques. Metrics adapted from the area of information extraction are also used to measure the effectiveness of the understanding techniques employed in this research.

The choice of a knowledge-base for program understanding is directly related to the program understanding approach followed. Since an approach using informal tokens was chosen, the most appropriate knowledge-base is a semantic network. This type of knowledge-base is appropriate since informal tokens are a form of natural language, and semantic networks are commonly employed for natural language processing. As this research demonstrates, the program understanding task at a certain level can be treated as a natural language understanding task. The semantic network employed by this approach differs from the semantic networks employed by earlier informal tokens approaches in that syntactical information is employed in the interface layer, and conceptual graphs are employed in the interior. The inferencing mechanism is similar to the inferencing mechanism of the semantic nets employed in earlier informal tokens approaches, but differs in that conceptual graphs as well as concepts can be used in the inferencing process. The use of conceptual graphs permits a deep, language-independent level of program understanding. Also, the employment of conceptual graphs in the semantic net allows the use of standard natural language generation techniques to produce a natural language description of the capabilities of a class.

3.3. Choices Made for the Reusability Analysis Approach

In deciding how to predict reuse, it became apparent that reuse must be considered at several levels. Four different views of reuse were devised: Reusability-in-the-Class, Reusability-in-the-Hierarchy, Reusability-in-the-Subsystem, and Reusability-in-the-Original-System. The reuse views Reusability-in -the-Class and Reusability-in-the-Original-System are used to determine whether a particular class is reusable. Reusability-in-the-Class examines particular aspects of reusability of a single class. Reusability-in-the-Original-System examines the usage of a class in the original system, which is a predictor of the reusability of the class in a new system.

Reusability-in-the-Hierarchy is used to determine whether a particular class hierarchy is reusable, and Reusability-in-the-Subsystem is used to determine whether a selected subsystem of classes is reusable.

These different views of reuse were measured using the following reusability quality factors: Modularity (which consists of cohesion and coupling), Interface, Documentation, and Simplicity (which consists of size and complexity). This selection of reuse quality factors provides a high-level, generic software quality metrics hierarchy, that will provide a good estimate of the reusability of a software component. They were adapted and modified from existing metrics using the rationale described below.

Two primary criteria required for a software component to be considered reusable are for the component to be easily understood and easily modified [21][85][86][24]. If a software component is not easily understood, it is unlikely to be chosen for reuse. A software engineer would likely choose a more easily understood component, even if that component were not implemented as well as another, simply because he would be more certain that the easily understood component provided the necessary functionality. Most reuse of components requires at least some modification of the components and thus the requirement for easy modifiability is important. The Documentation quality factor was chosen since documentation is very important to both the understandability and modifiability, and therefore the reusability, of software.

Code complexity is important from a software component reusability standpoint because it tends to predict understandability and modifiability. For example, a complex software component is hard to understand, and thus hard to modify. The size of a component is important in a similar way from a software reusability standpoint since larger components tend to be more complex and thus harder to understand and modify. The Simplicity quality factor was chosen to reflect the complexity and size of a software component as important factors in the understandability and modifiability, and thus the reusability, of software.

The interface of a software component is important from a reusability standpoint since the more understandable and clean the interface, the more easily the component can be integrated into a new system. The Interface quality factor was chosen to reflect these considerations.

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

Figure 3. Reusability-in-the-Class

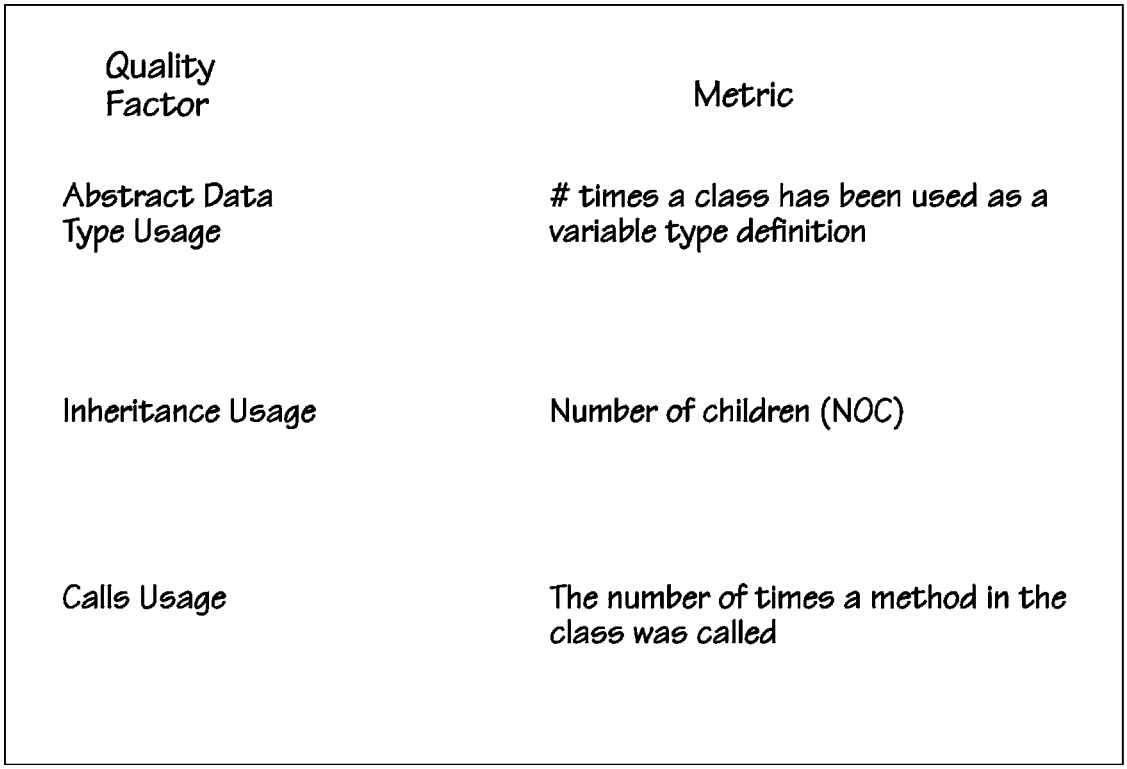


Figure 4. Reusability-in-the-Original-System

From the standpoint of a legacy software component, the modularity of the component is important. If the component is self-contained, is cohesive, and not overly coupled with other components, then it can easily be detached from its original system for reuse in another system. This also has applicability to the size of the component when reused in another system, e.g., if the component is uncohesive it can provide many functionalities, and only one might be needed in the new system and therefore the component requires storage for functionalities that are not used. A similar situation occurs if the component is heavily coupled with other components. The component itself might provide the needed functionality, whereas the other components provide additional, unused functionality, thus requiring storage for functionalities that are not used. The Modularity quality factor was chosen for these reasons.

The Interface, Documentation, and Simplicity quality factors are considered to be various aspects of understandability and modifiability. If the interface to a class is small and clean, then

the interface is both understandable and easily modifiable. If the documentation of a class, and a class' methods is informative, then the class is easily understood, and thus more easily modified than a poorly documented class. If the class and the class' methods are small and uncomplex, then the class is both more easily understood and more easily modified than for a larger, complex class. Modularity is considered to be a predictor of how easily the class can be detached from its original system, and to a certain extent, of how closely the class will match the requirements needed in a system in which it is to be reused (of how the class will not provide excessive, unneeded functionality).

Certain quality factors that are important in general are less important from a reusability standpoint. For example, whether or not a component is efficient is not particularly important in determining if that component is easily reusable. In certain areas of reuse, the determination of a component's efficiency might be of overwhelming importance; however, the efficiency of a software component would have to be compared to the efficiency of other software components within the context of the requirements of that particular system. The determination of whether a component can be easily reused is separate from the determination of whether a particular component is appropriate for a particular system.

The quality factor Modularity is similar to the Asdjodi-Mohadjer reuse quality factors (for cohesion) modularization, encapsulation, and abstraction, and (for coupling) independence of execution, and to the GPALS quality factors abstraction, cohesiveness, coupling, and modularity. The quality factor Interface corresponds to the Asdjodi-Mohadjer reuse quality factor Clear module interface/standardization of interfaces. The quality factor Documentation has some similarity with the Asdjodi-Mohadjer reuse quality factor Availability of parts and documentation and is similar to the GPALS quality factor Understandability. The quality factor Simplicity is similar to a criterion that Asdjodi-Mohadjer defined for evaluating the reusability of software. This Asdjodi-Mohadjer criterion asked if the complexity of the software is acceptable, and is similar to the GPALS quality factor Size.

In order to measure the quality factors mentioned above, several standard, widely-used object-oriented metrics have been used. The relationship between the metrics and the quality

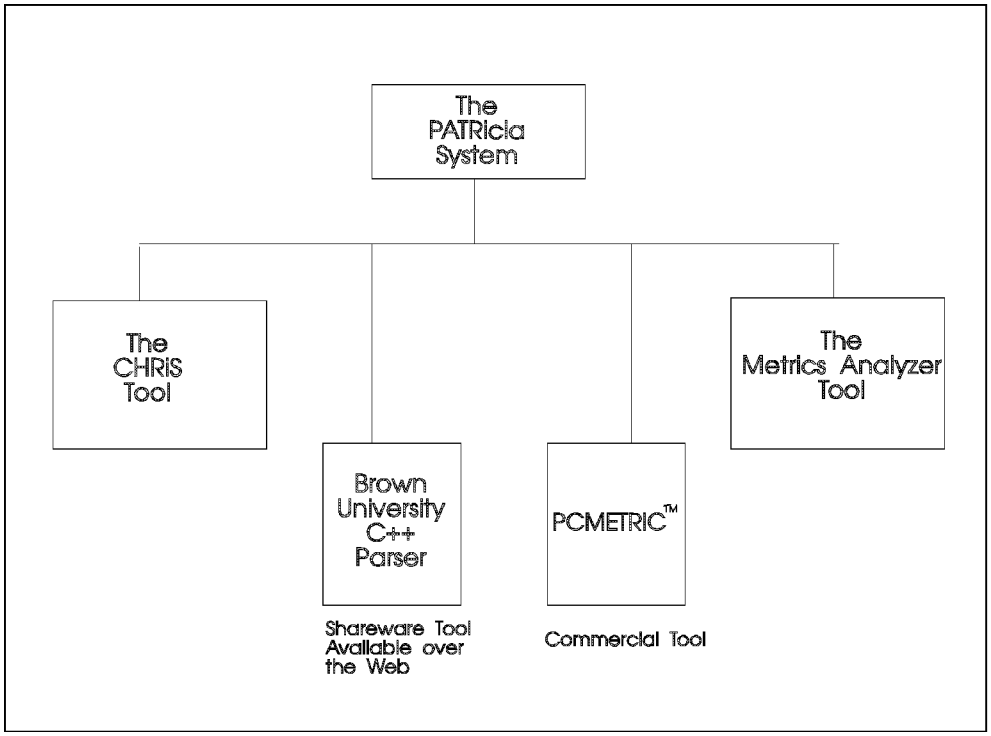


Figure 5. PATRicia System Structure Chart

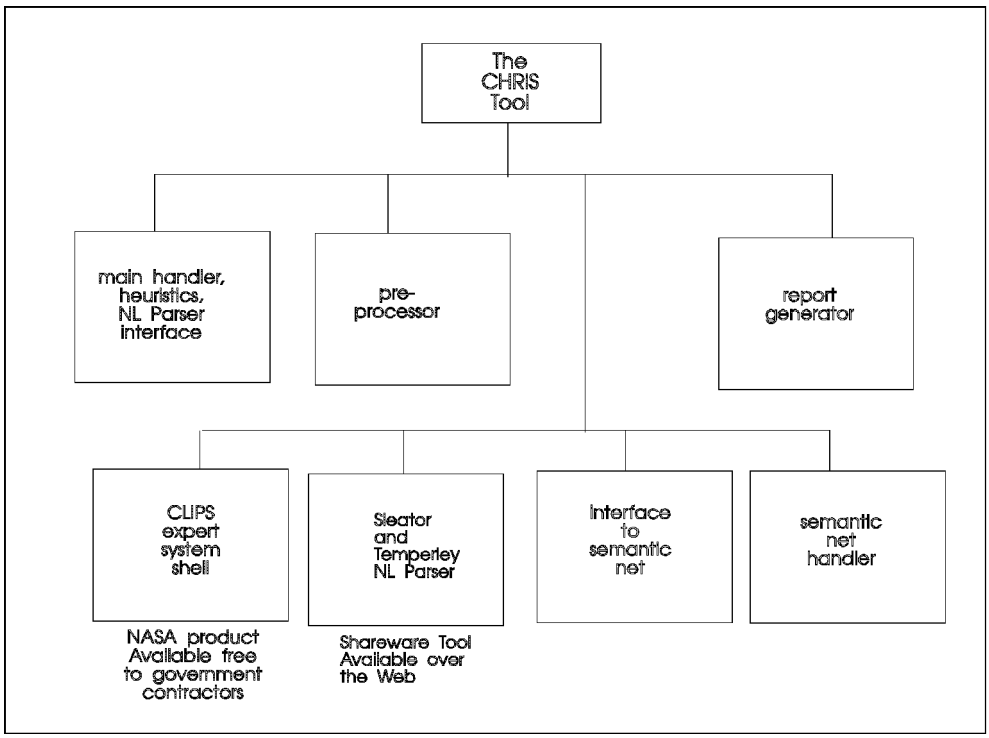


Figure 6. CHRiS Structure Chart

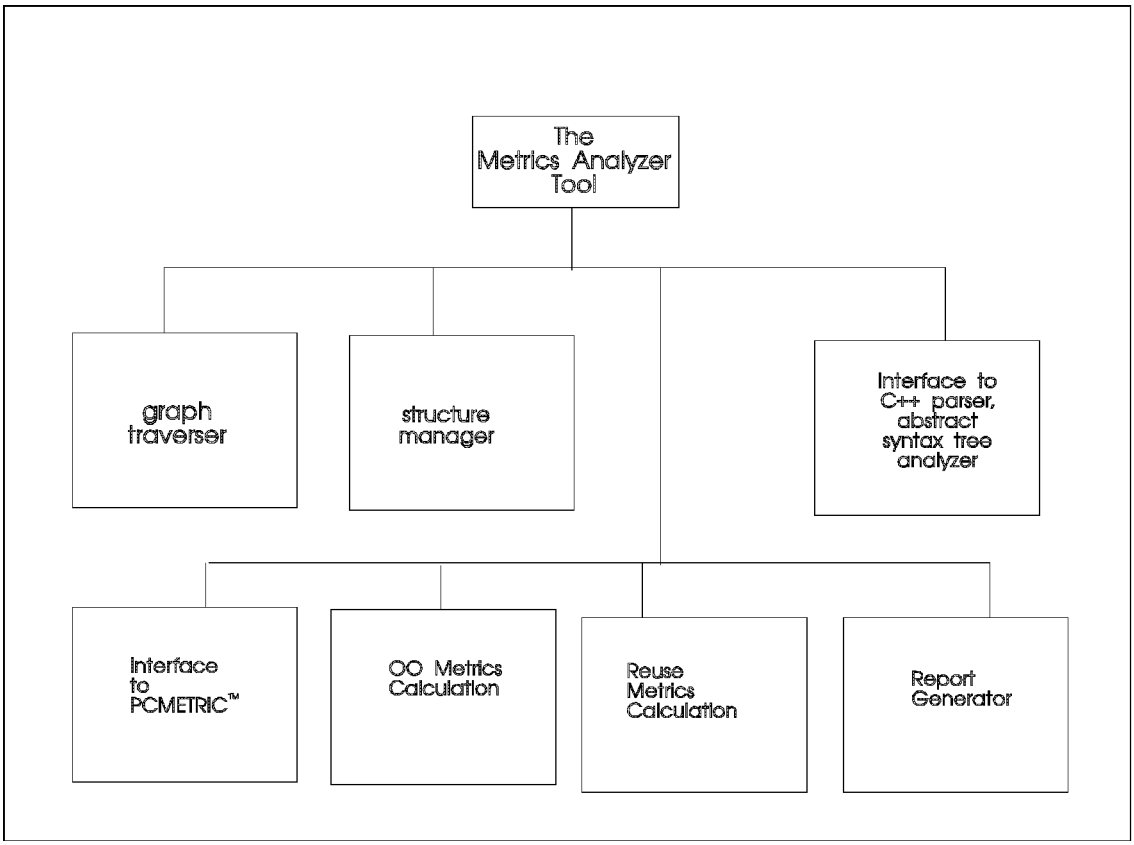


Figure 7. Metrics Analyzer Structure Chart

factors are shown in Figures 3 and 4. The quality factors and metrics used for Reusability-in-the-Hierarchy and Reusability-in-the-Subsystem are the same as those shown in Figure 3, but the metrics are collected over several classes instead of a single class. The detailed description of these metrics and their use in measuring quality factors is contained in Chapter V.

3.4. Choices Made for Automation

Automation of the understanding/identification and qualification approaches required the development of a tool (the **P**rogram **A**nalysis **T**ool for **R**euse, also known as the **PATR**icia system). The tool carries out the following functions: 1) syntactical analysis of natural language, 2) C++ parsing, 3) semantic analysis of natural language, 4) reusability quality metrics analysis,

and 5) object-oriented metrics collection. Since some of these functions are standard functions, but also functions that can be difficult and time-consuming to develop, it was decided to take advantage of existing software tools.

3.4.1 Tool Selection for the PATRicia System

The organization of the **PATR**icia system is shown in the structure charts in Figures 5, 6, and 7. The syntactical analysis of natural language is largely handled in the **PATR**icia system by the Sleator and Temperley natural language parser. The Sleator and Temperley parser was selected after an examination of several natural language parsers. The primary reasons for the selection of the Sleator and Temperley parser were: 1) it is a fast parser, $O(n^3)$, 2) it is a word-based parser that can easily be modified for sublanguage use (the use of sublanguages in the program understanding approach is described in section 4.2 below), 3) it is a widely-used natural language parser.

C++ parsing is handled in the **PATR**icia system by the Brown University C++ parser. The primary reasons for the choice of the Brown University C++ parser were: 1) it provides the output of a C++ program in the form of an abstract syntax tree stored in a text file, which can be easily processed, and 2) other similar projects in the computer science department had already successfully used the Brown University C++ parser, which indicated that it could be easily employed in another such project.

A tool used in the **PATR**icia system for the collection of object-oriented metrics is the PCMETRIC™ tool [89]. This inexpensive tool calculates some low-level object-oriented metrics. It was felt to be robust enough to calculate metrics needed for the operation of the **PATR**icia system.

The semantic analysis of natural language, the reusability quality metrics analysis, and some object-oriented metrics collection in the **PATR**icia system required new code written for those purposes, and will be described in Chapter VI.

3.5 Validating the Approaches

The validation of the research approach as implemented in the **PATR**icia system was done through analyzing the reusability of software in the domain of graphical user interfaces. This domain was chosen since it is large and complex enough to be a good test domain, and since there were many graphical user interface packages available to examine. There were two phases to the evaluation: 1) an evaluation of the program understanding phase of the **PATR**icia system, and 2) an evaluation of the reusability analysis phase of the **PATR**icia system.

The evaluation of the program understanding phase of the **PATR**icia system compared the understanding operation of the **PATR**icia system to that of human software developers. The human software developers agreed on a list of concepts that are handled by a particular C++ class. The **PATR**icia system then evaluated the class, and the concepts associated with the class by the **PATR**icia system were compared to the human software developers' list of concepts for the class. The system was then evaluated based upon the ability of the **PATR**icia system to identify those concepts that human software developers felt were contained in the class. The details of the evaluation of the program understanding phase of the **PATR**icia system is described in section 7.2.

The evaluation of the reusability analysis phase of the **PATR**icia system compared reusability decisions made by human evaluators to the reusability decisions made by the **PATR**icia system. Human evaluators evaluated the various reusability aspects of a class (or class hierarchy). The **PATR**icia system then evaluated the class (or class hierarchy), and the reusability values provided by the **PATR**icia system were compared to the human evaluators' reusability values. The system was evaluated upon its ability to predict reuse factors consistent with the evaluators' reusability values. A description of the reusability analysis phase of the **PATR**icia system is described in section 7.3.

Chapter IV

PROGRAM UNDERSTANDING APPROACH

4.1. Introduction

The program understanding approach employed in this research is a knowledge-based, heuristic approach that focuses on informal tokens from comments and identifiers. The approach employs natural language processing techniques for the analysis of comments and identifiers. A semantic network that is appropriate for natural language processing is used.

4.2. Description of Comment/Identifier Approach

4.2.1. The Use of a Sublanguage

The ability to treat comments as a sublanguage of English has a strong impact on the feasibility of the approach that was taken. It impacts the choice of the natural language parser, the dictionary provided for the natural language parser, and the size and complexity of the knowledge base.

A study of comments showed that comments can be considered to be a grammatical and subject-matter sublanguage of the English language. While this research focuses on comments written in the English language, the general approach is also appropriate for comments written in other languages. An additional study of identifiers in C++ classes showed that identifers tend to have certain common formats, and that by the use of heuristics involving these formats the

keywords extracted from the identifiers could be syntactically tagged. See Section 4.3 below for a description of both of these studies.

Based on these studies, it was determined that a natural language approach to the understanding of the comments and identifiers in a software component was appropriate to the understanding of that component. The natural language approach chosen employs separate syntactic and semantic phases. This is compositional semantic interpretation, in which semantic processing is applied to the parse of a sentence. Figure 8 demonstrates the interrelation of the various natural language processing phases.

4.2.2. The Syntactic Phase

In the syntactic phase, keywords from comments are syntactically tagged by the use of a natural language parser. Keywords from identifiers are syntactically tagged by the use of various heuristics based on common formats of identifiers. The operation of these heuristics also include the use of a natural language parser.

Selectional restraints within the natural language parser result in some removal of ambiguity during the syntactic phase. However, most ambiguity during this phase is removed by the use of a grammatically restricted language that is also restricted in subject-matter. The reduced subject matter of the sublanguage results in a restricted dictionary for the natural language parser, which limits the number of possible parses. The restricted grammar of the sublanguage also results in a restricted dictionary for the natural language parser since various word senses that are possible in unrestricted English are not included in the dictionary. This also limits the number of possible parses.

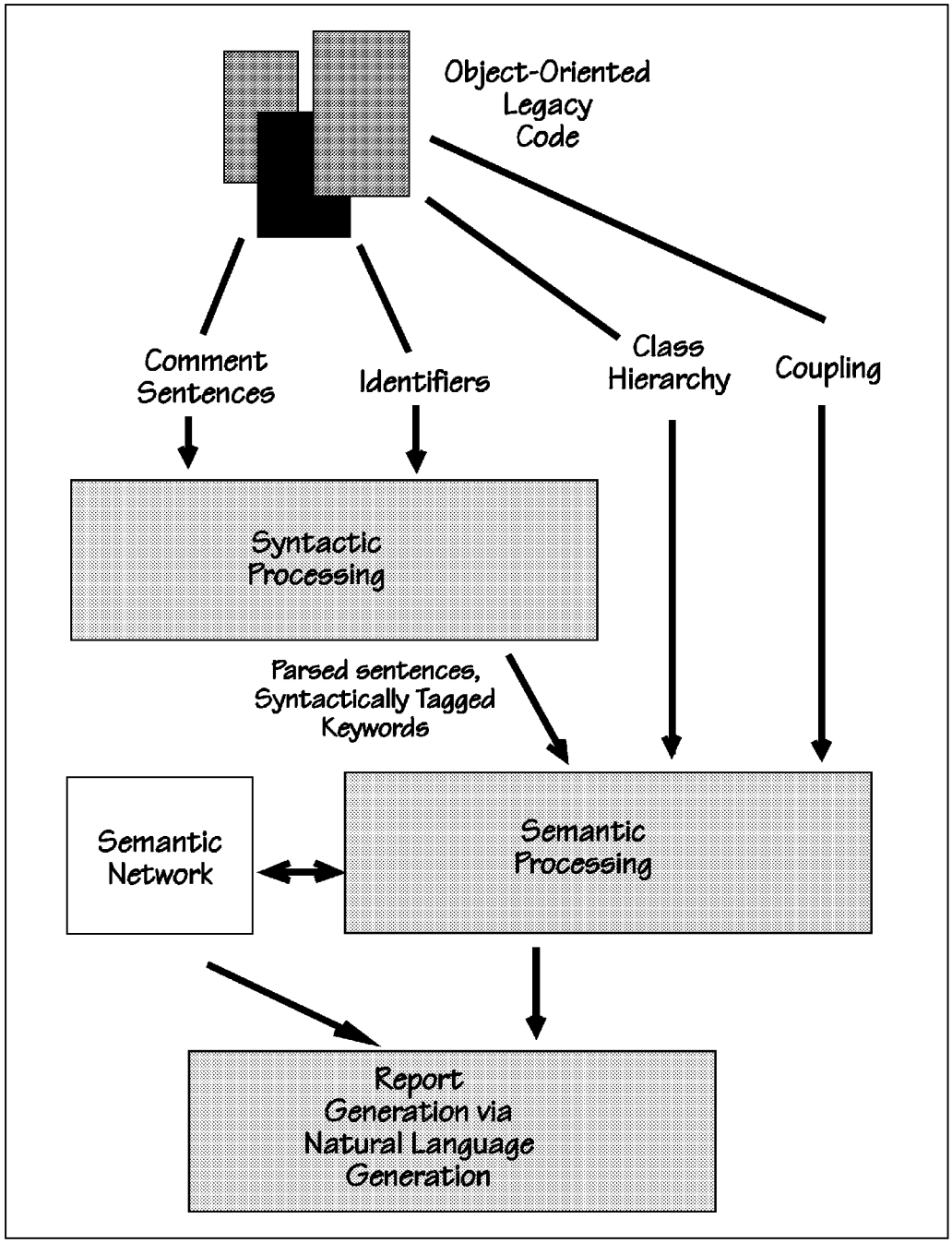


Figure 8. Natural Language Processing Phases in the Program Understanding Approach

4.2.3. The Semantic Phase

After the syntactic phase, the various parses are analyzed during the semantic phase of the program understanding approach. Additional ambiguity removal and final understanding of the software components are performed by the use of a knowledge-base in the form of a semantic network.

In the semantic phase, understanding follows the class hierarchy. The base classes are understood first, followed by derived classes. This choice of the class hierarchy as the interest set results in the class hierarchy performing the role of a "slice" in interactive program understanding tools. A "slice" in an interactive tool is a user-selectable interest set of variables and statements.

Two different types of information are desired as output from this approach. First, a list of standard concepts covered by a software component is required. This list of concepts can be used as criteria for the insertion of the class into a class library. Second, a natural language description of the operation of the class is required. This natural language description provides an indication of the functionality of the class from the point of view of the functionalities that are useful in the current domain. Both of these can be derived by use of the semantic net that is employed in understanding the software component.

4.2.4. The 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. Note that a keyword in the interface layer that is listed as an adjective is a separate keyword from a keyword of the same name in the interface layer that is listed as a noun, for example. 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. Figure 9 illustrates the semantic network employed in the program understanding approach.

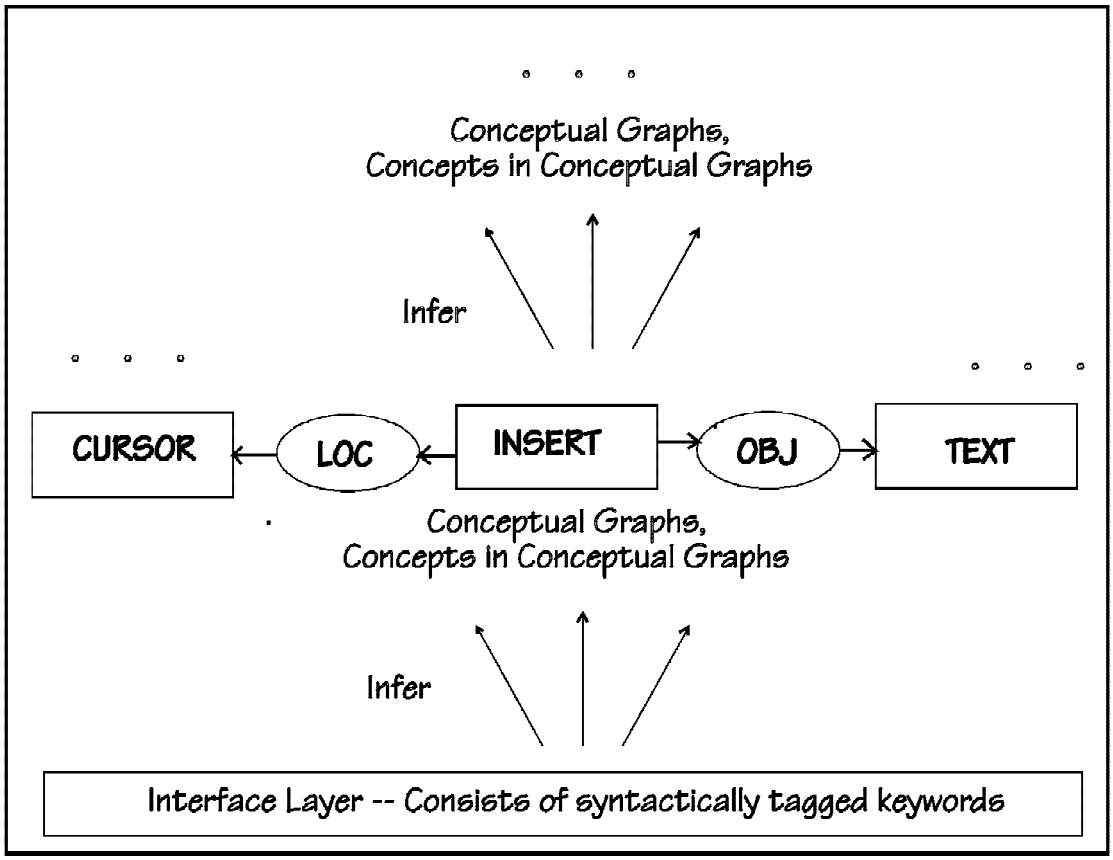


Figure 9. The Semantic Network Employed in The Program Understanding Approach

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 associated with it a weight. Each concept or conceptual graph in the interior has associated with it a threshold level. A concept "fires" when the weights of the links connecting to the concept achieve the weight of the threshold level.

The conceptual relations employed in the conceptual graphs within the semantic network include AGNT (for agent), OBJ (for object), ATTR (for attribute), LOC (for location), IS-A (this is a type relation), and IS_PART_OF (this is a whole-part relation).

IS-A relations and IS_PART_OF relations take part directly in inferencing. They have qualities similar to an inferencing link, as well as to a concept. If a concept with an IS-A relation

to a different concept is fired, then the different concept is also fired. Often that concept will have an IS-A link to yet another concept, and that concept is also fired. In other words, if a concept of a particular type is fired, then the type is fired, and passed on.

A concept with an "upward" (part to whole) IS_PART_OF relation to another concept will fire that concept only if several of the other parts of the concept are present (the weights on the inferencing IS_PART_OF links achieve the threshold value). A concept with a "downward" (whole to part) IS_PART_OF relation normally operates similarly to the IS-A relation in that the concept related to the current concept by the IS_PART_OF relation is also fired. However, it is also possible to have a threshold value attached to the whole to part relation. In some cases, when a high level concept is identified, one can assume definitely that the sub-parts of the concept are present, but in other cases the assumption is less strong. Consider a concept labeled "body of a human." One can strongly assume that the human has a heart and a brain that are related to the human being with a whole-part relationship. However, the assumption that a human possesses a pair of legs, arms, or eyes is less strong, since some human beings have lost one or more legs, arms, or eyes.

4.2.5. Natural Language Generation

Natural language generation is performed very simply in the program understanding approach. It is based on the low-level conceptual graphs in the semantic net. If an OBJ conceptual relation connects from concept A to concept B, then concept A is considered to be a verb, and concept B a noun that is the object of the concept A verb. If an AGNT conceptual relation connects from Concept A to Concept C, then Concept C is considered to be a noun, and the subject of the Concept A verb. In the case of a single AGNT link, the verb is normally considered to be singular (defaults to singular). If Concept A is connected to two or more concepts with AGNT conceptual relations, then those concepts are considered to form a compound subject, and the verb becomes plural. See Chapter VI for a discussion of how natural language generation has been implemented in the Program Analysis Tool for Reuse.

4.3. Feasibility Studies

A study of comments was performed to determine whether comments can be considered to be a grammatical and subject-matter sublanguage of the English language. An additional study of the syntactic format of identifiers in C++ classes was also performed.

4.3.1. Comments as a Sublanguage

In order to show that a particular corpus contains a sublanguage of natural language it is necessary to show that the language in the corpus is either grammatically restricted (a grammatical sublanguage) or restricted in semantic domain (a subject-matter sublanguage), or both. The grammatical restrictions in a sublanguage may involve commonality in the use of the imperative or the indicative, uniformity of tense, restriction of modality, and the deletion of articles (telegraphic). The restriction in semantic domain means that the language in the corpus will contain some words and word senses based on the intended use of the corpus, whereas other words and word senses will typically not be used.

An initial examination of typical comments led to the following observations:

1) Comments are usually written in the present tense, with either indicative mood or imperative mood. For example, "This routine reads the data." is present tense, indicative mood. "Read the data." is present tense, imperative mood.

2) The set of verbs typically used for comments is much restricted over the set of all English verbs. Verbs often used in comments include: is, uses, provides, implements, accesses, prints, inputs, outputs, reads, writes, supplies, defines, retrieves, gets, etc. Verbs seldom used in comments include: smiles, frowns, laughs, rides, flies, jumps, sings, fights, electrocutes, falls, punishes, hires, fires, pats, throws, pitches, calms, etc.

The set of comment verbs, while still very large, is still in general much smaller than the set of natural language verbs. Also, since the domain for analysis will also be restricted, the selection of verbs to support becomes much smaller.

3) Comments tend to be certain types. Two major types are header block comments and inline comments. Header block comments usually come in two formats--the operational description, and the definition. Examples of operational description header block comments are:

/* This routine reads the data */ and /* READ_DATA reads the data */

An example of a definition header block comment is:

/* GeneralMatrix--rectangular matrix class */

Header block comments, however, very often consist of several comment lines.

Inline comments also usually come in operational description and definition formats. An example of an operational description inline comment is:

// Get matrix row.

Examples of definition format inline comments include:

// index variable

// counter of incoming characters

Note that comments often have a telegraphic quality--certain determiners are left out. For example, a typical comment would be "Open file," rather than "Open the file." This telegraphic quality can be considered a "deviant" rule of grammar, that is common to a sublanguage.

The typical restriction of comments to present tense tended to indicate that comments could be considered as a grammatical sublanguage of English. The restriction on verbs tended to indicate that comments could be considered as a subject-matter sublanguage of English--particularly when considered as comments within computer code that is itself restricted to a limited domain.

Comments are either in sentence form, or not in sentence form. When in sentence form, it was found that comments are usually in either present tense, simple past tense, or simple future tense, but most often present tense. Other tenses were quite rare. A description of the common formats for sentence form comments is shown in Figure 10. When not in sentence form, it was found that comments still tended to have common formats. A description of the common formats for non-sentence form comments is shown in Figure 11. Comments were also examined as to content. A description of the common content of comments is shown in Figure 12.

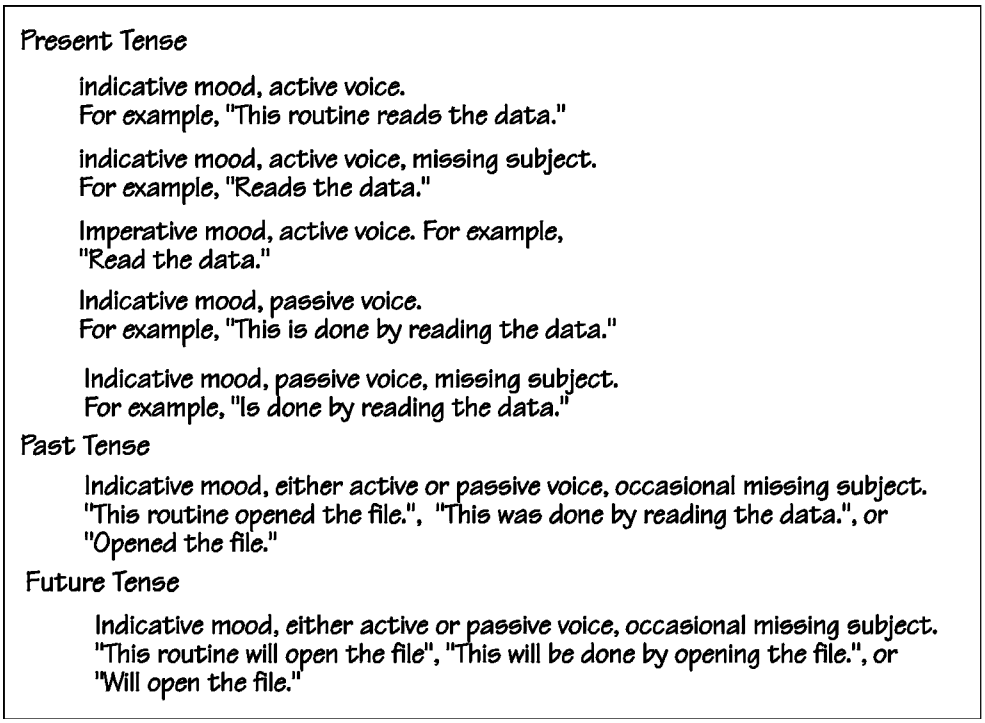


Figure 10. Common Formats of Sentence-Style Comments

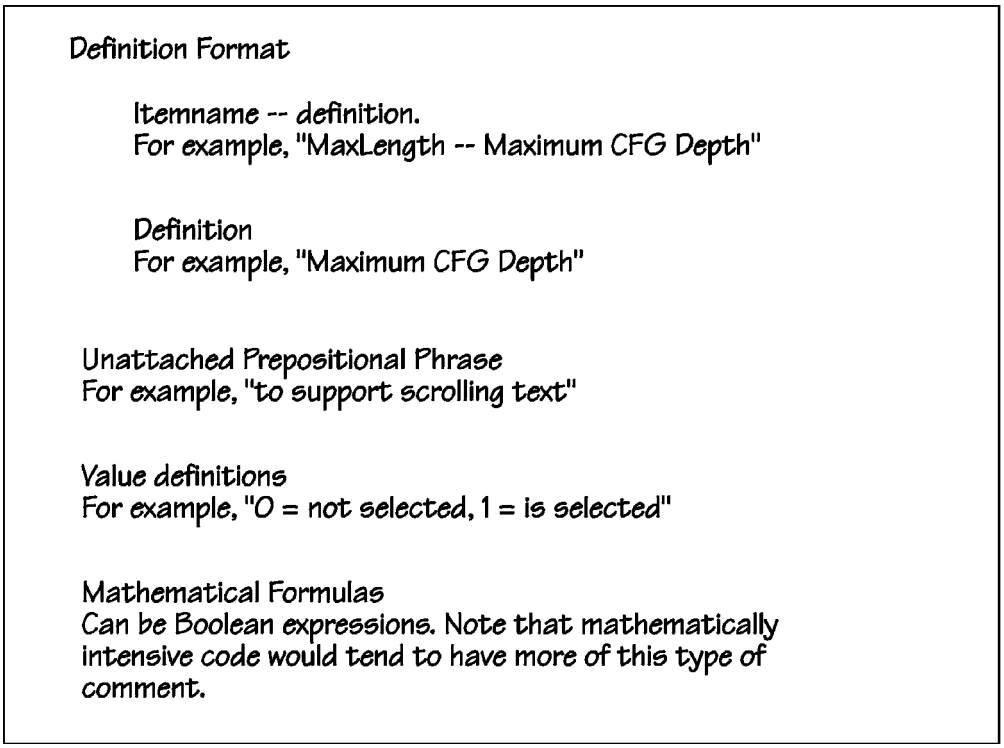


Figure 11. Common Formats of Non-sentence-Style Comments

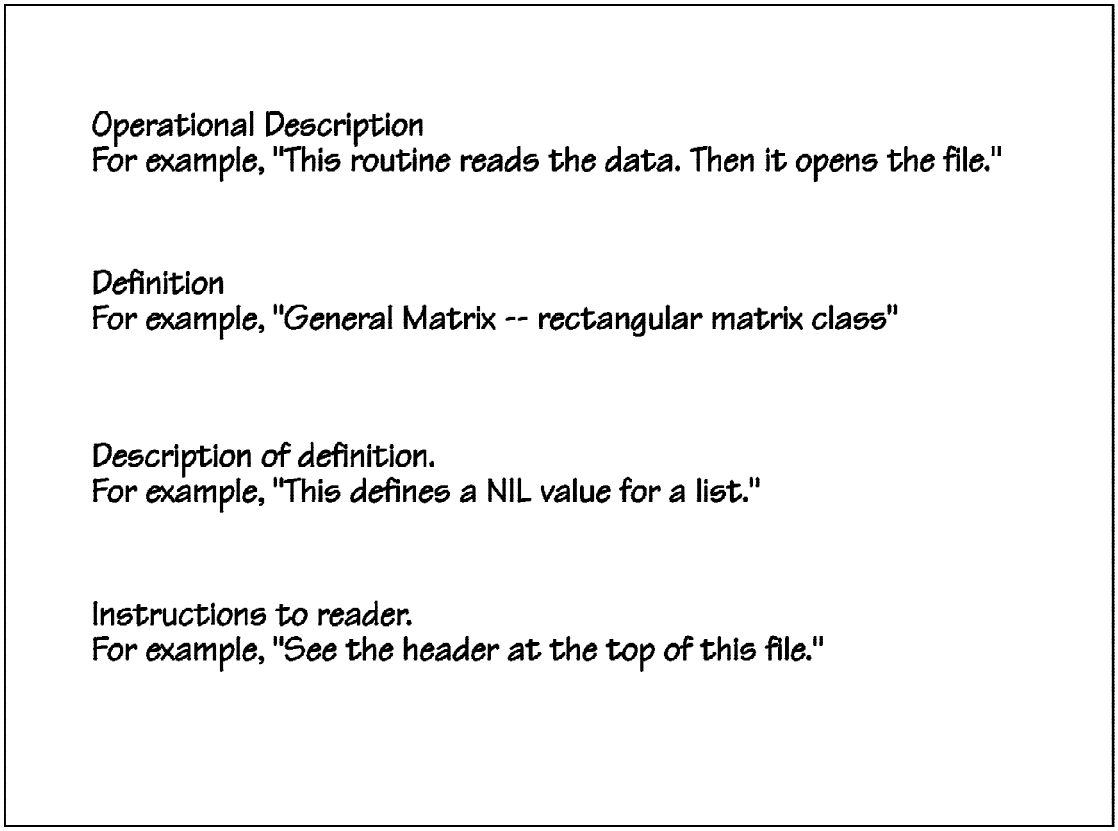


Figure 12. Common Content of Comments

The comment study was performed on three independent C++ graphical user interface packages [8][102][111], and a C++ parallelization tool [88]. A degree of randomness was achieved by examining comments from files selected by directory order. The first one hundred and eight comments in each package were examined, excluding revision and copyright notices. Typically this included comments from several different files. Comments that consisted of code were ignored, since it was felt that in this case the comment characters were performing two duties: one duty was to document the code, and the other duty was to remove code from the compilation process. This study was restricted to comments in C++ software. Figure 13 shows the results of an analysis of tense in sentence style comments. Note that 57% of all comments are in sentence style. From Figure 14, it can be seen that 82% of all sentence style comments are in present tense, and that 6% of sentence-style comments are not in present tense, simple future tense, or simple past tense. From Figure 15, it can be seen that 58% of present tense, sentence

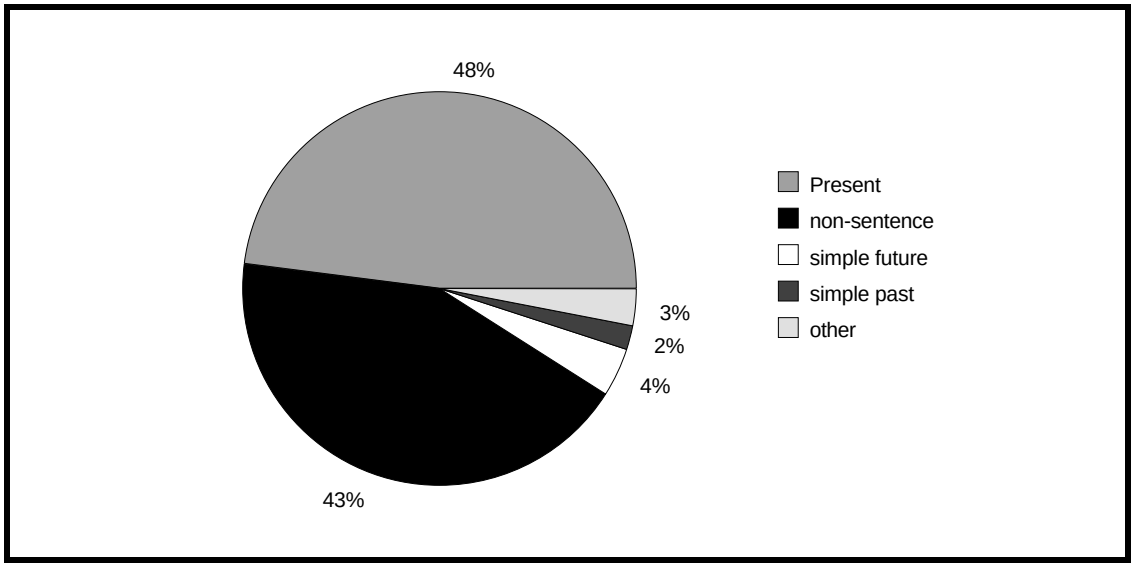


Figure 13. Sentence-style Comments versus Non-Sentence-style Comments

style comments are in indicative mood, active voice, while another 31% are in imperative mood, active voice. From Figure 16 it can be seen that definition format comments make up 90% of the non-sentence-style comments. These particular packages examined in the study had no comments of the Itemname--definition format, or of mathematical formulas. Figure 17 shows that 51% of the comments provide an operational description. Note that only 3% of all comments were not directly related to the immediate description of computer software. No instructions to the reader occurred in this study, although other packages that were examined did have a very small number of such comments.

From the above study, it can be seen that comments typically are grammatically restricated when compared to natural language as a whole, in relation to the use of tenses, mood, and voice employed. It can therefore be concluded that comments can be treated as a grammatical sublanguage of English. Since comments are used to describe the operation of computer software, and to define portions of computer software, comments can therefore also be treated as a subject-matter sublanguage related to the operation of computer software.

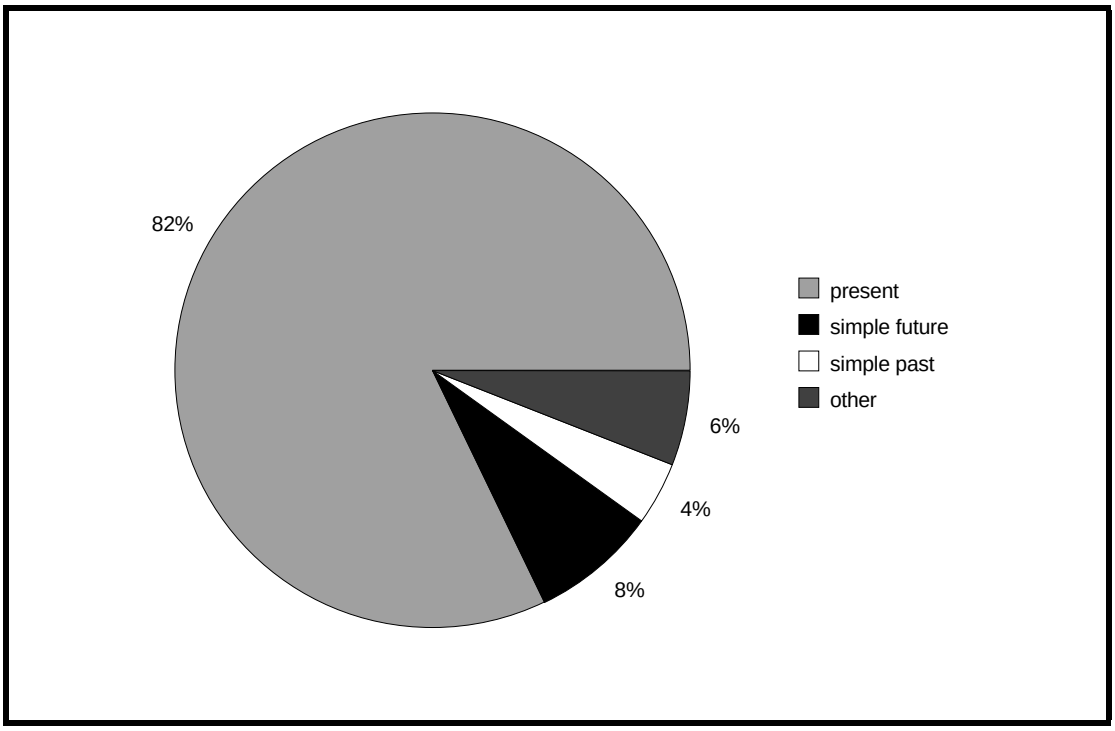


Figure 14. Analysis of Tense in Sentence-style Comments

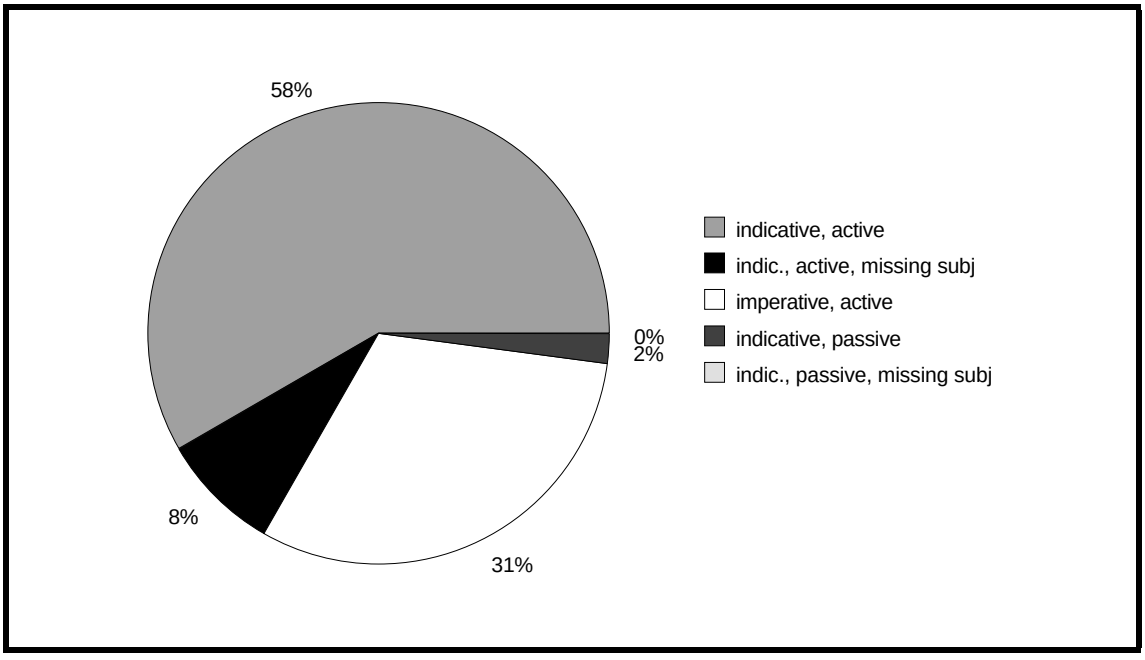


Figure 15. Mood and Voice of Present Tense Sentence-style Comments

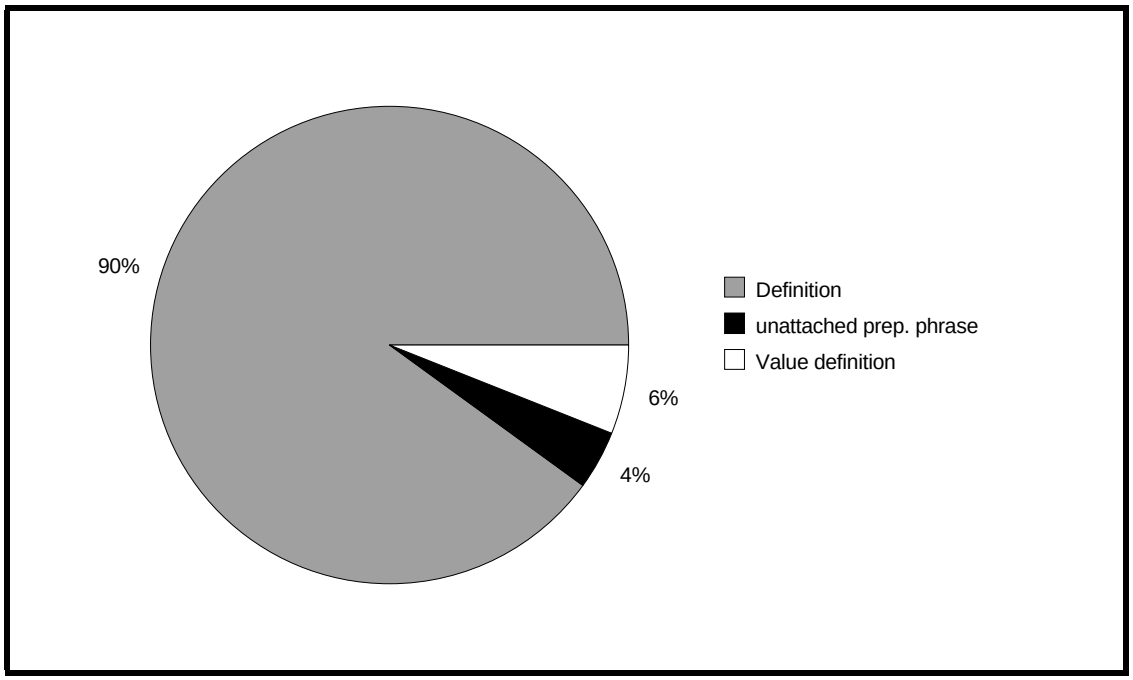


Figure 16. Non-sentence-style Comments

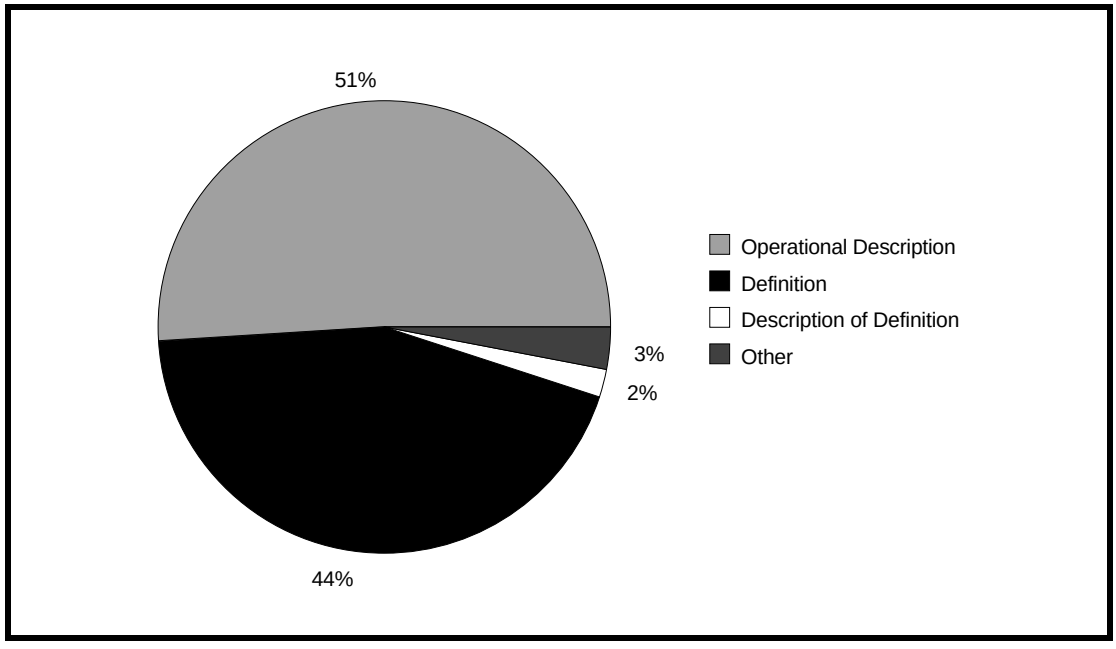


Figure 17. Content of Comments

4.3.2. Syntactic Tagging of Identifiers

The syntactic tagging of identifiers was approached by looking empirically at the syntactical format of C++ identifiers over 17 classes from 5 independent sources. These sources included three independent Graphical User Interface packages [8][102][111], a parallelization tool [88], and classes taken from a C++ textbook [112]. The results of this study are shown in Figures 18 and 19. Attribute (variable) identifiers are overwhelmingly noun related ({adj}* noun), whereas most member functions are verb related (either verb {adj}* noun [obj.] or {not+&}verb or verb combination). This seems reasonable when considered in regard to, for example, a rule of thumb for nomenclature in object-oriented systems given by Grady Booch [22]:

> "Modifier operations should be named with active verb phrases such as **draw** or **MoveLeft**. Selector operations should imply a query or be named with verbs of the form 'to be', such as **extentOf** or **isOpen**."

Consider also the following rules of thumb from Rumbaugh [99]:

> "Associations often correspond to stative verbs or verb phrases."
> "Attributes usually correspond to nouns followed by possessive phrases, such as 'the color of the car' or 'the position of the cursor.'"

An algorithm which can be used for the syntactic tagging of keywords extracted from member function identifiers is shown in Figure 20. This algorithm is based on the common formats of identifiers that were discovered during the identifier study. This algorithm utilizes a natural language parser in order to syntactically tag the keywords extracted from the identifiers. Some aspects of the algorithm such as the insertion of "the" after the first keyword in a two keyword identifier that is suspected of being an identifier with the format verb {adj}* noun[obj.] (two keywords, of course, means that no adjectives are present--an example two keyword identifier with this format would be "open_file"), or the insertion of "it" at the end of an identifier suspected to be of format {not+&}verb or verb combination (an example identifier with this format might be a single verb such as "open") are heuristics that attempt to make the sentence acceptable to the

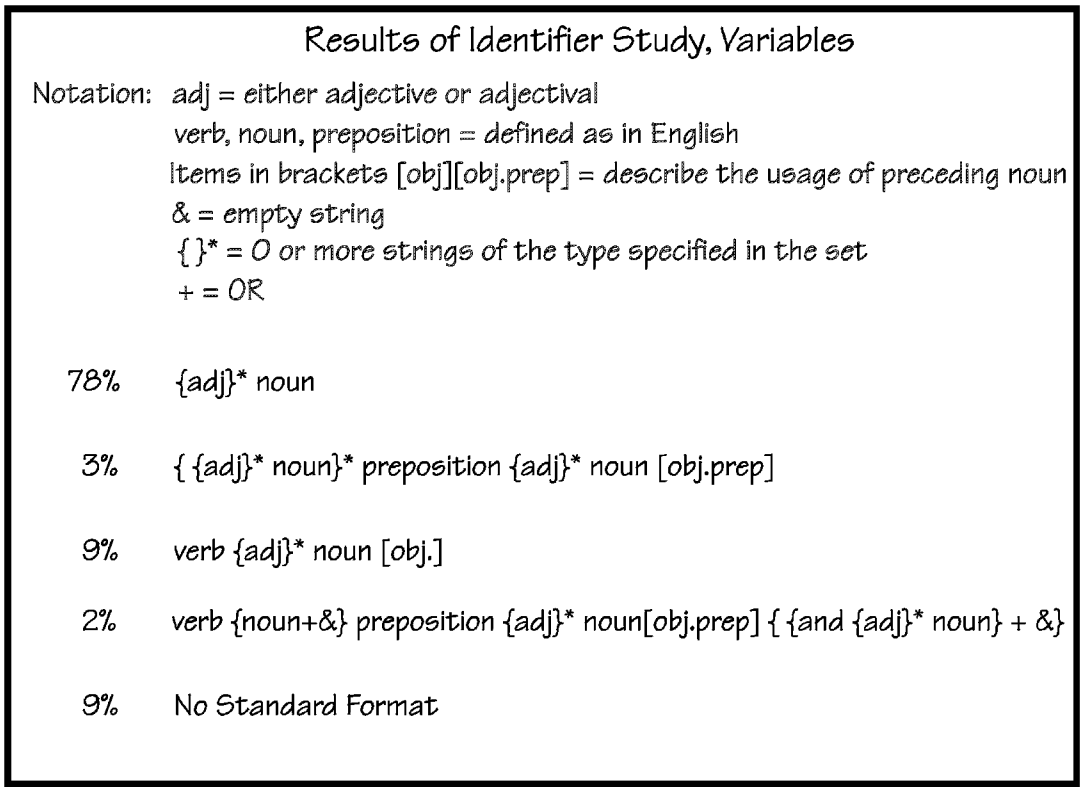


Figure 18. Results of Identifier Study, Variables

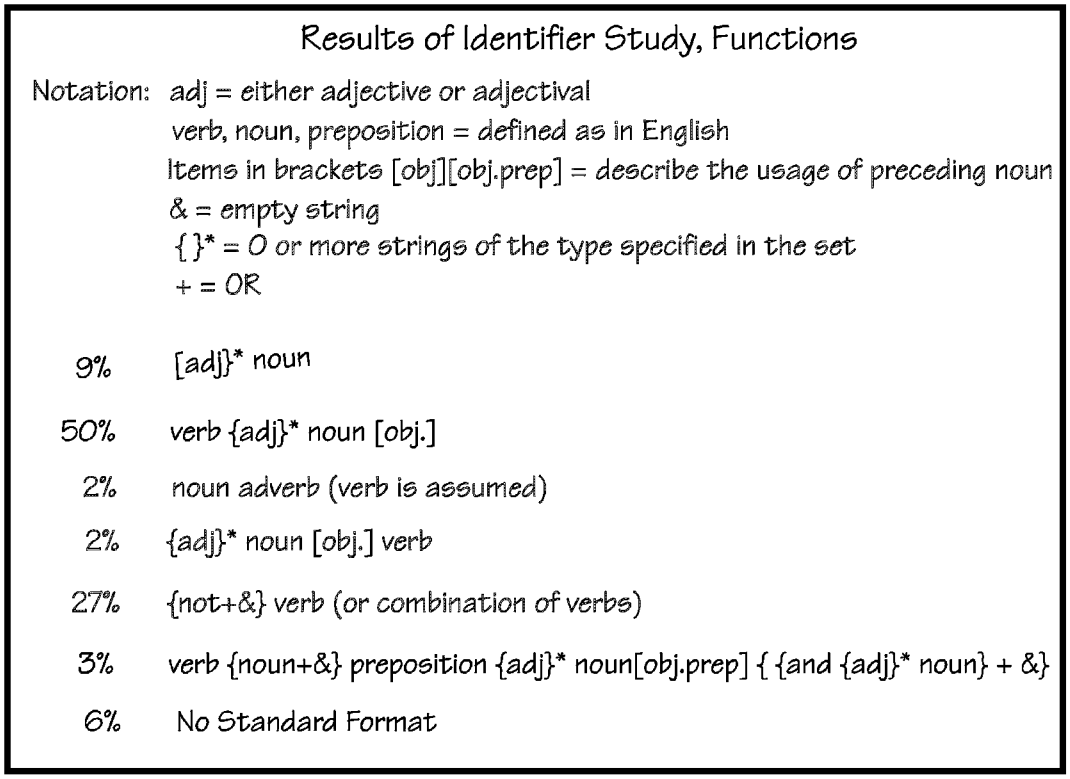


Figure 19. Results of Identifier Study, Functions

parser. Using the member function identifier algorithm shown in Figure 20, it should be clear that up to 82% of member function identifiers could be correctly syntactically tagged (those with formats verb {adj}* noun[obj.], {adj}*noun[obj.]verb, {not+&}verb or verb combination, verb {noun+&} preposition {adj}*noun{ {and {adj}*noun}+&}), depending upon the operation of the chosen natural language parser, whereas 18% of member function identifiers will not be syntactically tagged. However, the percentage of member function identifers that are not syntactically tagged would normally be rejected by the natural language parser, and would therefore not be tagged erroneously.

Attribute (variable) identifiers can be treated in a similar manner. However, the attribute algorithm shown in Figure 21 results in the erroneous tagging of a small percentage of identifiers. This algorithm correctly handles all {adj}* noun, { {adj}* noun}* preposition {adj}* noun [obj.prep], and most verb {adj}* noun [obj.] and verb {noun + &} preposition {adj}* noun { {and {adj}* noun} + &} identifiers. In the best case this would be 92% of identifiers. However, all No Standard Format identifiers would be erroneously tagged as {adj}* noun. For that reason, the attribute algorithm was not employed in this approach. Note that the attribute algorithm could be improved by heuristics similar to those used for the function algorithm.

```
Capitalize first keyword of identifier.
Put a period at end of the identifier.
Apply identifier to natural language (NL) parser.
If the identifier is accepted as a legal sentence by the NL parser then
  (probably verb {adj}* noun[obj.] or
  verb {noun+&} prep. {adj}* noun[obj.prep] { {and {adj}* noun} +&} )
  Use those sentence usage results returned by the NL parser
Else
  If two keywords in identifier then
    Insert "the" after the first keyword, try again
    If the new sentence is accepted as a legal sentence then
      (probably verb {adj}* noun[obj.] )
      use those sentence usage results returned
      by the NL parser
    Else
      reverse those words (looking for {adj}* noun [obj.] verb )
      apply new sentence to NL parser
      If the new sentence is accepted as a legal
      sentence then
        use those sentence usage results returned by the
        NL parser
      Else
        Insert "the" after the first keyword, try again
        If the new sentence is accepted then
          use those sentence usage results returned by the NL parser
        Else
          assert keywords separately, no usage information
Else (not 2 keywords in identifier)
  (looking for {not+&} verb or verb combination )
  Add "it" as extra keyword of sentence (at end of original identifier)
  Apply new sentence to NL parser
  If the new sentence is accepted as a legal
  sentence then
      use those sentence usage results returned by
      the NL parser
  Else
      assert keywords separately, no usage information
```

Figure 20. Algorithm for Syntactic Tagging of Member Function Identifiers

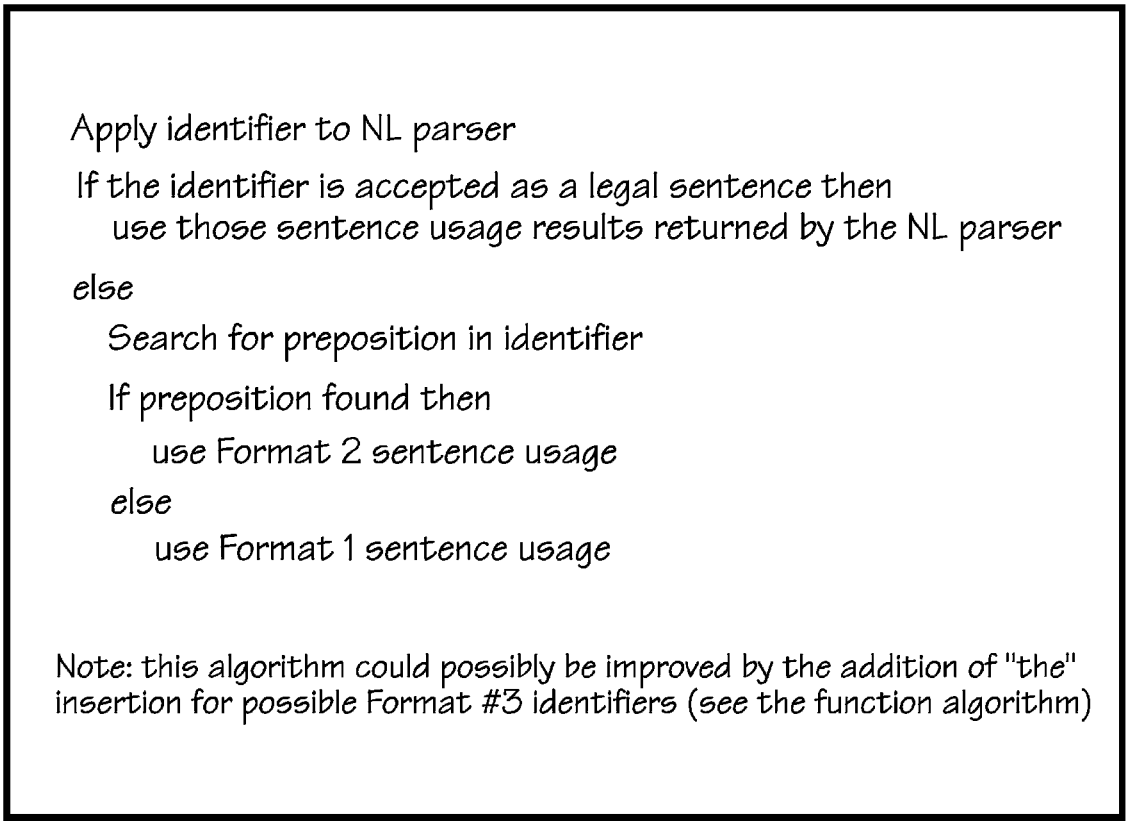


Figure 21. Algorithm for Syntactic Tagging of Attributes (Variables)

Chapter V

REUSABILITY ANALYSIS APPROACH

5.1. Introduction

Reusability of software components in object-oriented software can be examined at various levels, depending on what set of software components might be chosen for reuse, and how individual software components compare to all software components that must be reused along with the individual software components. These reuse views can be considered as software quality factor frameworks [80][92][98]. They differ from generalized software quality factor frameworks in that they do not attempt to derive an overall look at quality, but are limited to quality factors that affect reuse.

The following reuse views of an object-oriented system have been defined: Reusability-in-the-class, Reusability-in-the-hierarchy, Reusability-in-the-original-system, and Reusability-in-the-subsystem.

5.2. Reusability Software Quality Metrics Hierarchies

5.2.1. Reusability-in-the-Class

Reusability-in-the-Class consists of qualities of an individual class that tend to make a class reusable. Figure 22 lists the various quality factors, subfactors, and metrics that together form the Reusability-in-the-Class view. Modularity is a measure of the independence and

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

Figure 22. Reusability-in-the-Class

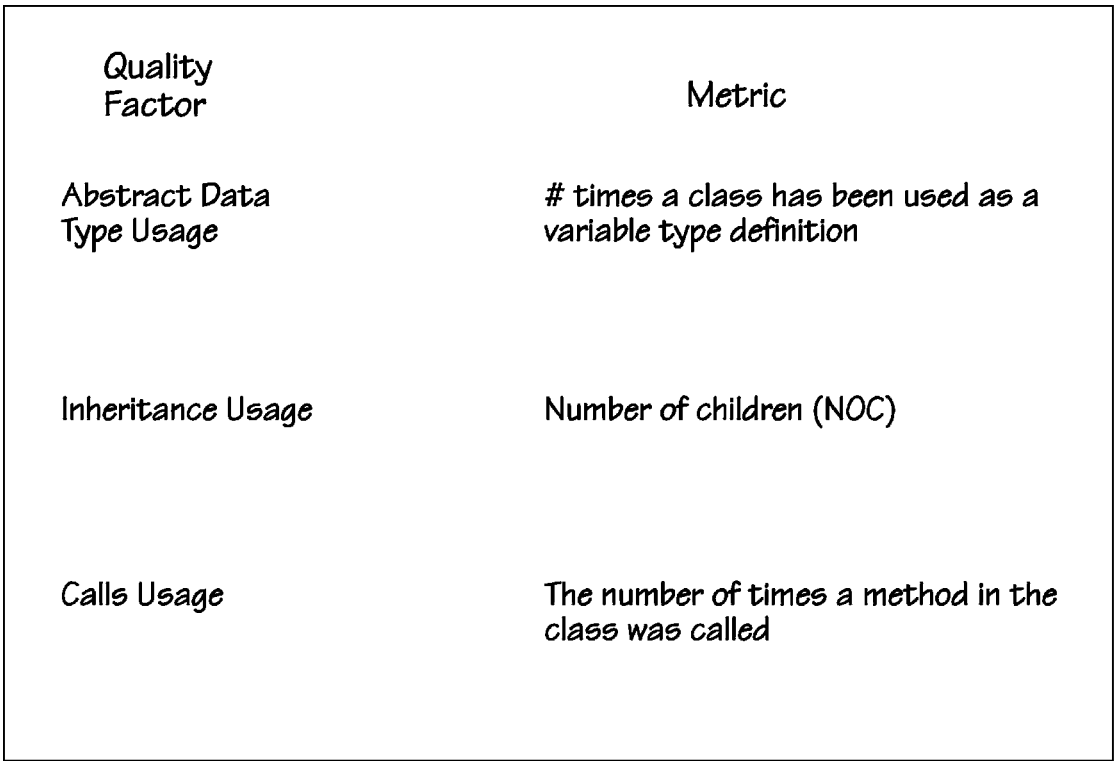


Figure 23. Reusability-in-the-Original-System

self-containment of the class. Interface complexity measures how hard the class is to integrate into a new system. Interface Complexity, Documentation, and Simplicity together form an indication from various viewpoints of the class' understandability and modifiability. Reusability-in-the-class is calculated as a weighted sum of its quality factors. During validation studies, all quality factors were weighted equally.

5.2.2. Reusability-in-the-Original-System

The number of uses of a class in the original system for which it was designed can be used as a predictor of reuse in a new system. Figure 23 lists the quality factors and metrics that together form the Reusability-in-the-Original-System view. The three different quality factors, Abstract

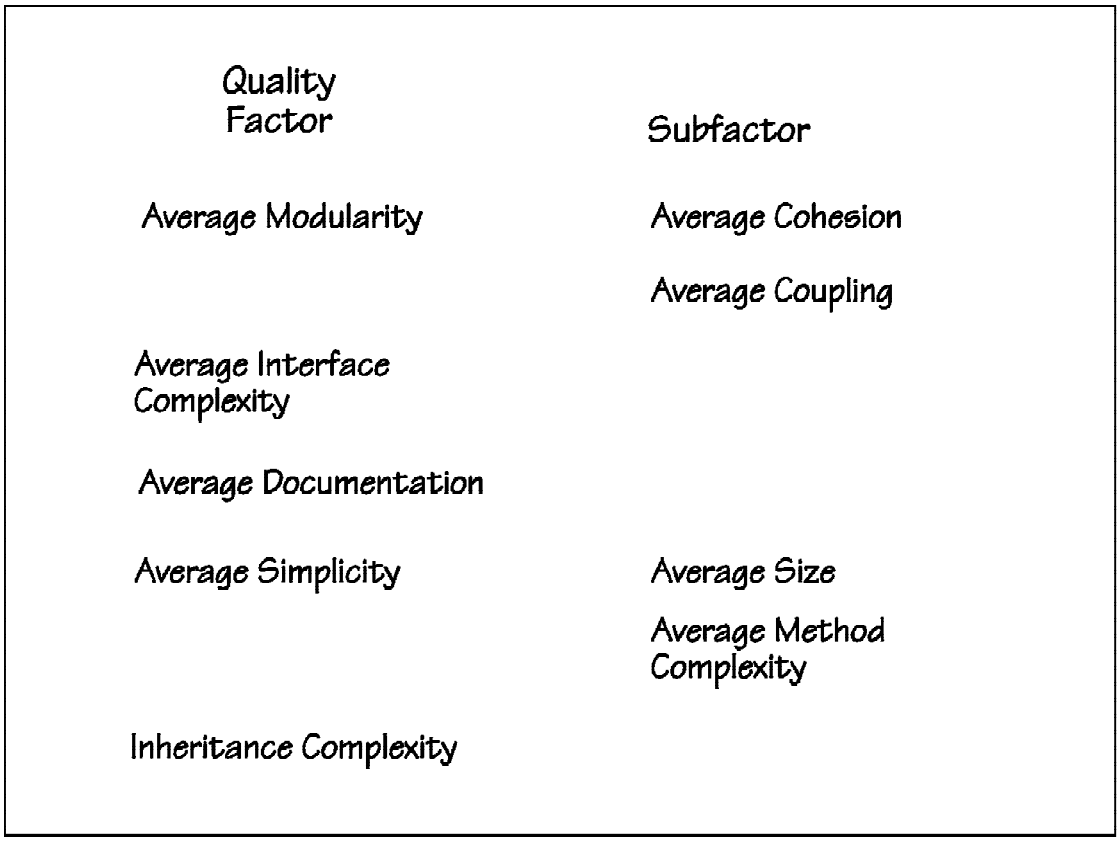


Figure 24. Reusability-in-the-Hierarchy and Reusability-in-the-Subsystem Quality Factors Hierarchy

Data Type Usage, Inheritance Usage, and Calls Usage correspond to three different ways that a class can be used. A class can be used as a type definition for variables (Abstract Data Type Usage); it can be inherited by a derived class that can reuse its functionality and add additional functionality (Inheritance Usage); and various member functions within the class can be called (Calls Usage). Reusability-in-the-Original-System is calculated as a weighted sum of its quality factors. Currently, all quality factors were weighted equally.

5.2.3. 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 all the reusability quality factors of all the classes in the inheritance hierarchy. Qualities of the inheritance hierarchy, primarily the depth of the inheritance tree, tend to affect the reusability of classes in a particular class hierarchy. Figure 24 lists the various quality factors and subfactors that form the Reusability-in-the-Hierarchy view.

Reusability-in-the-Hierarchy is calculated as a weighted sum of the quality factors Average Modularity, Average Interface Complexity, Average Documentation, Average Simplicity, and Inheritance Complexity. During validation studies, quality factors were weighted equally. Average Modularity is calculated as a weighted sum of Average Cohesion and Average Coupling. Average Simplicity is calculated as a weighted sum of Average Size and Average Method Complexity. During validation studies, quality subfactors were weighted equally.

5.2.4. Reusability-in-the-Subsystem

A selected subsystem of classes might provide a particular functionality that is useful in a new system. The reusability of such a subsystem is a sum of the qualities related to reuse for all individual classes in the subsystem. In addition, all inheritance hierarchies in the subsystem would tend to influence reuse of the subsystem.

The quality factor hierarchy for Reusability-in-the-Subsystem would be the same as for Reusability-in-the-Hierarchy. However, the set of classes for which Average Cohesion, Average Coupling, Average Interface Complexity, Average Documentation, Average Size and Average Complexity are calculated would be different. Also, Inheritance Complexity would require a somewhat different calculation, which is described in Section 5.3 below.

5.3. Object-oriented Metrics to Predict the Reusability Quality Factors

In this section, the various object-oriented metrics that have been chosen to predict the various reusability quality factors are described, and the rationale behind the choice of each metric is discussed.

5.3.1. Reusability-in-the-Class

5.3.1.1. Modularity

Modularity, as specified in the Reusability-in-the-Class hierarchy, consists of Cohesion and Coupling. As values for Coupling increase, Modularity tends to decrease. Various object-oriented metrics are used to predict Cohesion and Coupling.

5.3.1.1.1. Coupling

The Coupling quality factor is measured by a weighted sum of the number of friend functions, the number of external variable accesses, and the number of message sends (Message Passing Coupling, or MPC). The number of friend functions, the number of external variable accesses, and the number of message sends are used by Lorenz and Kidd [78]. The number of message sends, otherwise known as Message Passing Coupling (MPC) was first defined by Li and Henry [75].

The Coupling quality factor is non-inheritance-related Coupling. Non-inheritance-related Coupling is undesirable coupling, and a high value for Coupling is undesirable.

Friend functions are a reference to an external function, and thus a form of coupling. They also tend to break the encapsulation of objects [78], and thus are undesirable from a Cohesion standpoint; however, if the metric had been used to measure Cohesion as well as Coupling, then it

would have applied twice to the Modularity calculation, which seemed inappropriate. Thus the decision was made to have the number of friend functions apply only to the Coupling calculation.

The number of external variable accesses, and the number of message sends are both references to entities that are defined outside the class hierarchy, and thus are a form of non-inheritance related coupling. External variable accesses includes both global variables, and public variables in another class. The number of message sends is calculated by Li and Henry [75] as equivalent to the number of external function calls. Thus calls to global functions are included in the number of message sends, as are calls to public member functions in other classes.

Other coupling metrics such as Chidamber and Kemerer's Coupling Between Objects (CBO) and Rajaraman and Lyu's Class Non-inheritance-related-coupling (CNIC) were considered, but were not chosen for several reasons.

The CBO metric was not chosen for use for the following reasons: 1) it combines two metrics used elsewhere, the number of message sends and the number of external variable accesses, which are themselves simpler metrics, and 2) it defines the message sends and instance variable accesses as occurring only to another class. Within the context of C++, all message sends (all external function accesses, including global functions as well as member functions in other classes), and all external variable accesses (global variables plus variables in other functions) should be included in a coupling calculation.

Rajaraman and Lyu defined two non-inheritance-related coupling metrics, Class Non-inheritance-related coupling (CNIC) and Class Coupling (CC). Class Non-inheritance-related coupling counted the accesses to variables and the uses of member functions that have neither been defined in the class itself, or in any of its proper ancestors. Similarly to the Chidamber and Kemerer CBO metric, this metric restricted the message passing calculation to member functions of other classes, which is not appropriate for C++.

Rajaraman and Lyu's Class Coupling metric is the sum of the number of global variable references, global function calls, references to instance variables in other classes, and number of messages to other classes. This metric is exactly equivalent to the Coupling quality factor--it was

felt that this combination of low level metrics is more appropriate considered as a quality factor than simply as a higher level metric.

Lorenz and Kidd [78] provide empirical threshold values based on many C++ and Smalltalk projects, and on their own observations of what is appropriate for object-oriented software, for the number of message sends, and the number of friend functions.

5.3.1.1.2. Cohesion

The object-oriented metric that is normally considered when trying to measure cohesion is the Lack of Cohesion of Methods (LCOM) metric. The multiple definitions of the LCOM metric currently in use include: 1) the original definition of the LCOM metric by Chidamber and Kemerer [27][48], 2) the definition of the LCOM metric provided by Li and Henry [74][75][76][48], 3) the redefinition of the Li and Henry version of the LCOM metric by Hitz and Montazeri [58][48], and 4) the redefinition of their original LCOM metric by Chidamber and Kemerer [26][48][49].

Chidamber and Kemerer originally defined the Lack of Cohesion in Methods metric (LCOM) as the number of disjoint sets of methods of a class formed by the intersection of the set of instance variables accessed by each method [27]. This definition of disjointness of sets was somewhat ambiguous, and was further defined by Li and Henry [74][75]:

> "LCOM = number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to N; where N is a positive integer."

Hitz and Montazeri proposed a different, graph-theoretic formulation of Li and Henry's version of the LCOM metric, which they claim is equivalent to Li and Henry's definition of LCOM.

Chidamber and Kemerer redefined the LCOM metric [26]:

"Consider a Class C_1 with n methods M_1, M_2, ..., M_n. Let $\{ I_j \}$ = set of instance variables used by method M_i. There are n such sets $\{ I_1 \}$ ... $\{ I_n \}$. Let $P = \{ (I_i, I_j) \mid I_i$ intersecting I_j is equal to the null set $\}$ and $Q = \{ (I_i, I_j) \mid I_i$ intersecting I_j is not equal the null set $\}$. If all n sets $\{ I_1 \}...\{ I_n \}$ are the null set then let P = the null set.
LCOM = $|P| - |Q|$, if $|P| > |Q|$

= 0 otherwise"

In this definition, Q = the number of pairs of methods which both access the same instance variables. P = the number of pairs of methods which do not have any instance variable accesses in common.

This definition of LCOM has some drawbacks in that classes with widely different cohesions will result in the same LCOM value [11][58]. Chidamber and Kemerer themselves note that "the LCOM metric for a class where $|P| = |Q|$ will be zero. This does not imply maximal cohesiveness, since within the set of classes with LCOM = 0, some may be more cohesive than others" [26].

There are some variations on the LCOM metric that are independent of which definition is used. Some of these variations include the determination of exactly which member variables and member functions take part in the calculations. One question of LCOM implementation is whether or not inherited member variables should be used as part of the cohesiveness determination. Neither of the Chidamber and Kemerer definitions specifies whether or not inherited variables should be used. The Li and Henry definition specifies local variables only. Another possible problem with the implementation of the LCOM metric is the inclusion of the constructor member function and/or the destructor member function in the LCOM metrics calculation. Constructor member functions, for example, tend to include all or most of the member variables. This is reasonable, since a primary purpose of a constructor function is to initialize the member variables of a class. However, including the constructor in an LCOM calculation can result in an LCOM value that erroneously measures a class with poor cohesion as having good cohesion. See Figure 25 for an example of LCOM implementation with constructor and without constructor.

Since the different definitions and implementations of the LCOM metric can result in different values for LCOM, the question is which definition and which implementation of LCOM should be chosen. An experiment to determine which LCOM definition and implementation best measures cohesion was performed as part of this research. In this experiment, C++ classes and hierarchies of classes were chosen from three independent C++ graphical user interface (GUI) packages. Seven experienced domain experts rated each class for cohesiveness by categorizing it as acceptably cohesive, or not cohesive. An acceptably cohesive class was classed as 100%, a non-cohesive class as 0%.

LCOM was measured for these classes in 8 different ways: 1) revised Chidamber and Kemerer definition, including inherited variables, including constructor function, 2) revised Chidamber and Kemerer definition, including inherited variables, not including constructor function, 3) revised Chidamber and Kemerer definition, not including inherited variables, including constructor function, 4) revised Chidamber and Kemerer definition, not including inherited variables, not including constructor function, 5) Li and Henry definition, including inherited variables, including constructor function, 6) Li and Henry definition, including inherited variables, not including constructor function, 7) Li and Henry definition, not including inherited variables, including constructor function, and 8) Li and Henry definition, not including inherited variables, not including constructor function.

The various LCOM values collected for each class are shown in Figure 26. There are certain interesting aspects to these results. First, consider class number 7. This class results in large values for both the Chidamber and Kemerer revised definition, and the Li and Henry definition, so presumably the class possesses a large number of non-cohesive member functions. However, consider the values for the Chidamber and Kemerer LCOM definition. The largest number occurs in the implementation of LCOM that does not use inheritance, and that does include the constructor function. In this case the value is 1142. This value is well over twice as large as the next largest value for that implementation, which is 435. This occurs because the Chidamber and Kemerer revised definition is bounded by the number of combinations of two functions possible in the number of member functions (n choose 2), which is:

```
Class example  {
      int Visible;
      int X,Y;
      float salary;
      int val_from_port;

      example( );  // Constructor
      void Hide( );
      void calculate_salary(int num_months,
                        float amt_per_month);
      void read_val_from_input_port( );
}

void example::example( ) {
       Visible = false;
       salary = 0.0;
       val_from_port = 0;
}

void example::Hide( ) {
       Visible = false;
        putpixel(X,Y, getbkcolor() );
}

void calculate_salary(int num_months, float
                        amt_per_month) {
       salary = num_months *
amt_per_month;
}

void read_val_from_input_port( ) {
       val_from_port = inportb(PORTA);
}
```

Using the revised Chidamber and Kemerer metric,
P = 3, Q = 3, so LCOM = |P| - |Q| = 0, which supposedly means the class is completely cohesive

Using the revised Chidamber and Kemerer metric, but leaving out the constructor,
P = 3, Q = 0, so LCOM = |P| - |Q| = 3, which is a more reasonable cohesiveness value, since the class is actually not cohesive

Using the Li and Henry metric,
the LCOM sets are { example( ) (constructor), Hide( ), calculate_salary( ), read_val_from_input_port( ) }
so LCOM = number of disjoint sets = 1, which supposedly means the class is completely cohesive

Using the Li and Henry metric, but leaving out the constructor,
the LCOM sets are: {Hide( ) }
{calculate_salary( ) }
{read_val_from_input_port ( ) }
so LCOM = number of disjoint sets = 3, which is a more reasonable cohesiveness value, since the class is actually not cohesive

Figure 25. LCOM Implemented With and Without Constructor

n! / [(2!) * (n-2)!] = n(n-1)/2

This can result in large values of LCOM for very non-cohesive functions with large numbers of member variables, whereas most other classes will have very much smaller values of LCOM.

Now consider classes 8 and 9. When inheritance is included in the calculation, class 8 is considered to be more cohesive than class 9 by both the Chidamber and Kemerer revised definition and the Li and Henry definition, irregardless of whether or not the constructor function is included in the calculation. However, when inheritance is not considered in the calculation, class 8 shows as less cohesive than class 9. Similarly for classes 13 and 14.

Using the Li and Henry definition of LCOM, no changes in rank of cohesiveness of classes were found when comparing implementations that did not consider the constructor function versus those that did consider the constructor function, although in several cases the relative distance between values in a comparison of one class versus another did vary.

When considering the Chidamber and Kemerer definition of LCOM, a change in rank of cohesiveness was found when comparing implementations that did not consider the constructor function versus those that did consider the constructor function. This change in rank is in classes 14 and 15. With the constructor function, class 14 shows as more cohesive than class 15. Without the constructor function, class 14 shows as less cohesive than class 15.

A linear regression study was performed that compared the experts' ratings of cohesiveness to the various implementations of the LCOM metric. This was a simple study with LCOM as the independent variable, and cohesiveness (as measured by the experts) as the dependent variable. For the Li and Henry definition of LCOM, the best results were obtained using the LCOM implementation that did not include inheritance, and that did include the constructor function. In this case the regression was highly significant ($p < 0.0001$), and the R^2 value was 66%. R^2 for the Li and Henry definition, without inheritance and without the constructor was 62%. R^2 for the Li and Henry definition, with inheritance, was 49% with the constructor, and 46% without the constructor. A scatter plot of the Li and Henry definition, without inheritance, and with constructor, is shown in Figure 27.

	Chidamber and Kemerer revised definition				Li and Henry definition			
Class	with Inheritance with Const.	with Inheritance without Const.	without Inheritance with Const.	without Inheritance without Const.	with Inheritance with Const.	with Inheritance without Const.	without Inheritance with Const.	without Inheritance without Const.
1	34	34	34	34	8	8	8	8
2	24	16	24	16	6	5	6	5
3	343	321	343	321	24	24	24	24
4	287	276	287	276	18	18	18	18
5	247	247	247	247	21	21	21	21
6	1	0	1	0	2	1	2	1
7	1142	1118	1142	1118	37	44	37	44
8	195	166	435	406	15	14	30	29
9	321	298	323	298	24	24	25	24
10	64	48	136	120	9	8	17	16
11	22	15	28	15	6	6	8	6
12	48	31	66	49	7	6	10	9
13	76	66	76	66	10	10	10	10
14	25	32	91	78	4	5	14	13
15	28	21	28	21	8	7	8	7
16	378	351	378	351	27	26	27	26
17	13	10	13	10	5	5	5	5
18	2	1	4	3	2	2	3	3

Figure 26. Different Calculations of LCOM Over 18 Different Classes

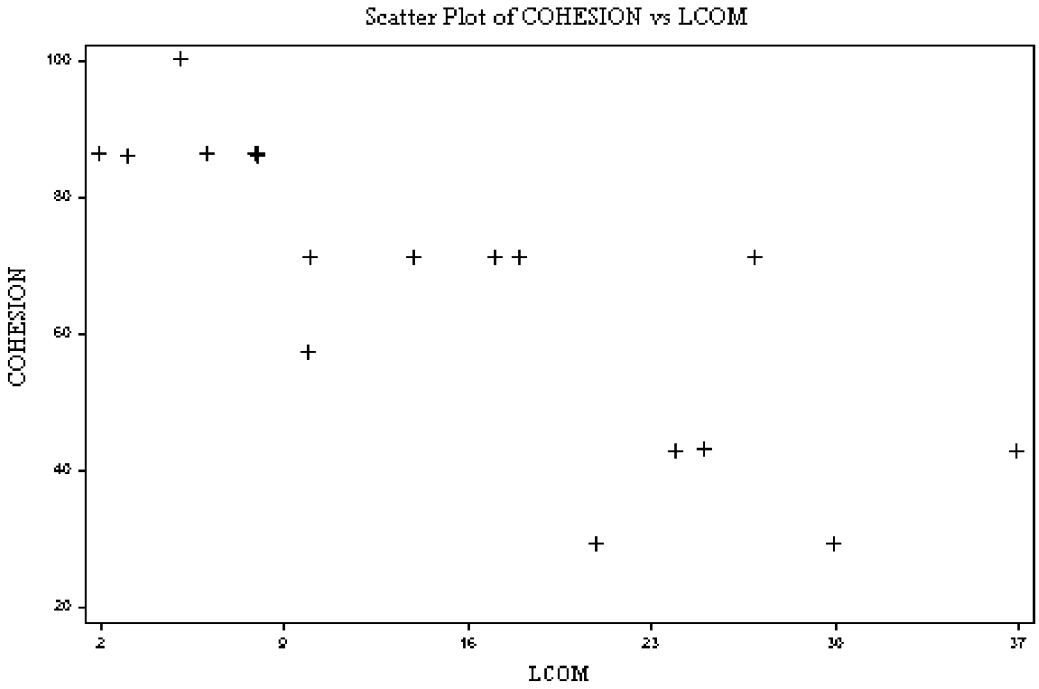


Figure 27. Li and Henry LCOM, Without Inheritance, With Constructor

Considering only the Li and Henry definition, these results were somewhat surprising in view of the anomalies discussed earlier relating changes in rank between classes measured with inheritance considered, and without inheritance considered. A possible reason for this might be that inherited variables are used as general purpose variables such as global display flags. The use of such variables would not be considered by the experts as showing a relationship between member functions. The difference between the Li and Henry implementations with the constructor, and without the constructor was very small.

For the revised Chidamber and Kemerer definition, the best results were obtained using the LCOM implementation that did not include inheritance, and that did include the constructor, although the difference from the Chidamber and Kemerer measured without inheritance but also without the constructor was minimal. However, the Chidamber and Kemerer metric had an R^2 only of 41% in the best case ($p < 0.0043$). R^2 was 30% in the worst case (with inheritance, without constructor). A scatter plot of the best case (without inheritance, with constructor) is shown in Figure 28.

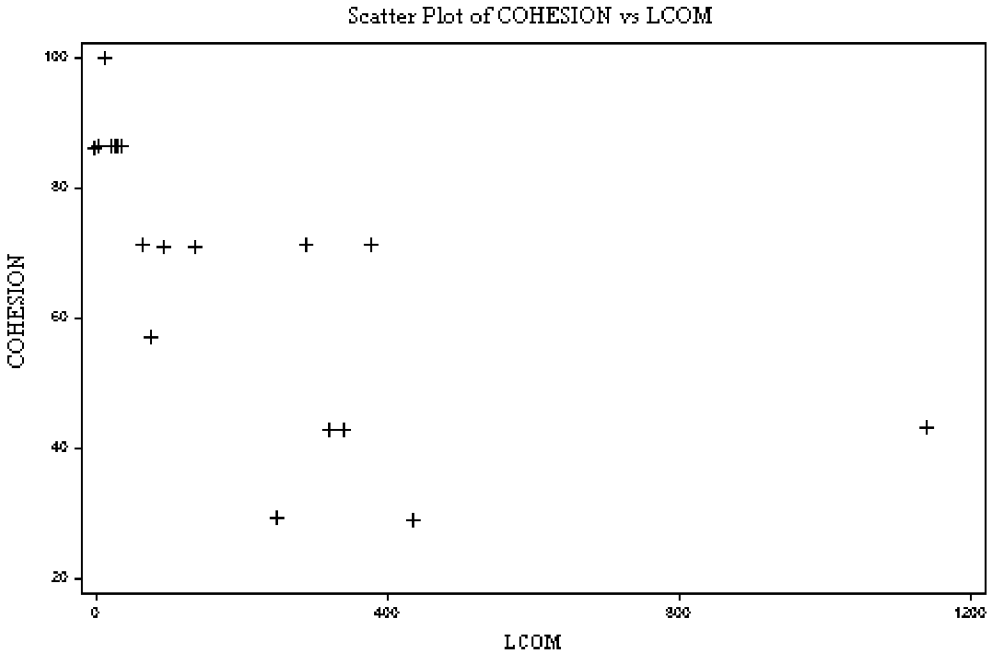


Figure 28. Chidamber and Kemerer LCOM, Without Inheritance, With Constructor

The implementation of LCOM that seemed to best measure cohesiveness was the Li and Henry definition of LCOM, which did not include inherited variables, and that did include the constructor function in the calculations, although the difference between this implementation of LCOM and the implementation of LCOM which did not include inherited variables and did not include the constructor function was small. Thus the LCOM definition and implementation that was chosen for use in the reusability hierarchy to measure the Cohesion quality factor was the Li and Henry definition of LCOM, implemented without inheritance, and with the constructor function.

5.3.1.2. Interface Complexity

Li and Henry [74][75] used the number of local methods to measure the size of the interface of a class. Lorenz and Kidd used the number of public instance methods in a class as a measure

of the services that are available from that class to other classes [78]. Lorenz and Kidd provided an empirical threshold value for this metric based on a large number of C++ and Smalltalk projects.

Thus Interface Complexity is measured by the number of public methods metric based on the Lorenz and Kidd metric. This was chosen since the rationale behind the Lorenz and Kidd metric seemed appropriate--that the interface of a class with respect to other classes is best described by the services provided by the class. The Li and Henry metric, the number of local methods, includes private and protected methods as well as public methods. Private methods are not really part of the interface of the class, since they are not accessible external to the class.

5.3.1.3. Documentation

Lorenz and Kidd used the Average Number of Comment Lines per Method metric and the Average Number of Commented Methods metric as indicators of the good implementation of a class [78]. They provided empirical threshold values for both these metrics based on a large number of C++ and Smalltalk projects. Additionally, it was felt that comments were also necessary in the class definition. Thus a count of comments compared to the number of attributes and member functions is also used in the calculation of Documentation. Documentation is calculated as a weighted sum of all three of these metrics. During validation studies, the weights were equal.

5.3.1.4. Simplicity

Simplicity, as specified in the Reusability-in-the-Class hierarchy, consists of Size and Complexity of Methods. As Size and Complexity of Methods increase, Simplicity tends to decrease. Numerous object-oriented metrics are used to predict Size and Complexity.

5.3.1.4.1. Size

Lorenz and Kidd [78] calculated size using the number of methods in a class, the number of attributes in the class, and the average method size in Lines of Code. They also provided empirical thresholds for the metrics based on a large number of C++ and Smalltalk projects. Li and Henry [78] also used two similar metrics: SIZE1, the number of semicolons in a class, and SIZE2, the number of attributes in a class plus the number of local methods. Size is calculated here using the number of methods in a class, the number of attributes in the class, and the average method size in Lines of Code. This is the same as those metrics used by Lorenz and Kidd, and similar to those used by Li and Henry.

5.3.1.4.2. Complexity

The complexity of a class is measured using the Weighted Methods per Class (WMC) metric, that was originally defined by Chidamber and Kemerer in their 1991 OOPSLA paper [27]. The definition of the Weighted Methods per Class (WMC) metric remained basically the same in Chidamber and Kemerer's 1994 IEEE Transactions on Software Engineering paper [26]. In both papers, WMC is the sum of the complexities of all local methods in the class. In both cases, the definition says "If all method complexities are considered to be unity, then WMC = n, the number of methods."

Neither of the definitions specifies how the complexity of the methods is to be determined. However, Li and Henry [75] specifies that the measure of the complexity of a single method is McCabe's cyclomatic complexity metric. Thus Li and Henry calculated WMC as the sum of the McCabe's cyclomatic complexities of each local method. Complexity here will be calculated using Li and Henry's definition of WMC.

5.3.2. Reusability-in-the-Original-System

Reusability-in-the-Original-System is calculated as a weighted sum of three different quality factors, Abstract Data Type Usage, Inheritance Usage, and Calls Usage. Abstract Data Type usage is the number of times the class was used as a type definition for a variable. This has some similarity to the Li and Henry metric Data Abstraction Coupling (DAC), except that the purpose of the metric is different. Inheritance Usage is calculated as the Number of Children (NOC) of a class. Calls Usage is the number of calls to member functions in the class.

5.3.3. Reusability-in-the-Hierarchy

Reusability-in-the-Hierarchy is calculated as a weighted sum of the quality factors Average Modularity, Average Interface Complexity, Average Documentation, Average Simplicity, and Inheritance Complexity (see Figure 24). During validation studies, each quality factor was weighted equally. Average Modularity is calculated as a weighted sum of Average Cohesion and Average Coupling. Average Simplicity is calculated as a weighted sum of Average Size and Average Method Complexity. During validation studies, each subfactor was weighted equally. Inheritance Complexity is the depth of the inheritance tree.

5.3.4. Reusability-in-the-Subsystem

Reusability-in-the-Subsystem is calculated the same as Reusability-in-the-Hierarchy, except that the set of classes over which it is calculated is different, and Inheritance Complexity is the maximum depth of any hierarchy tree in the subsystem.

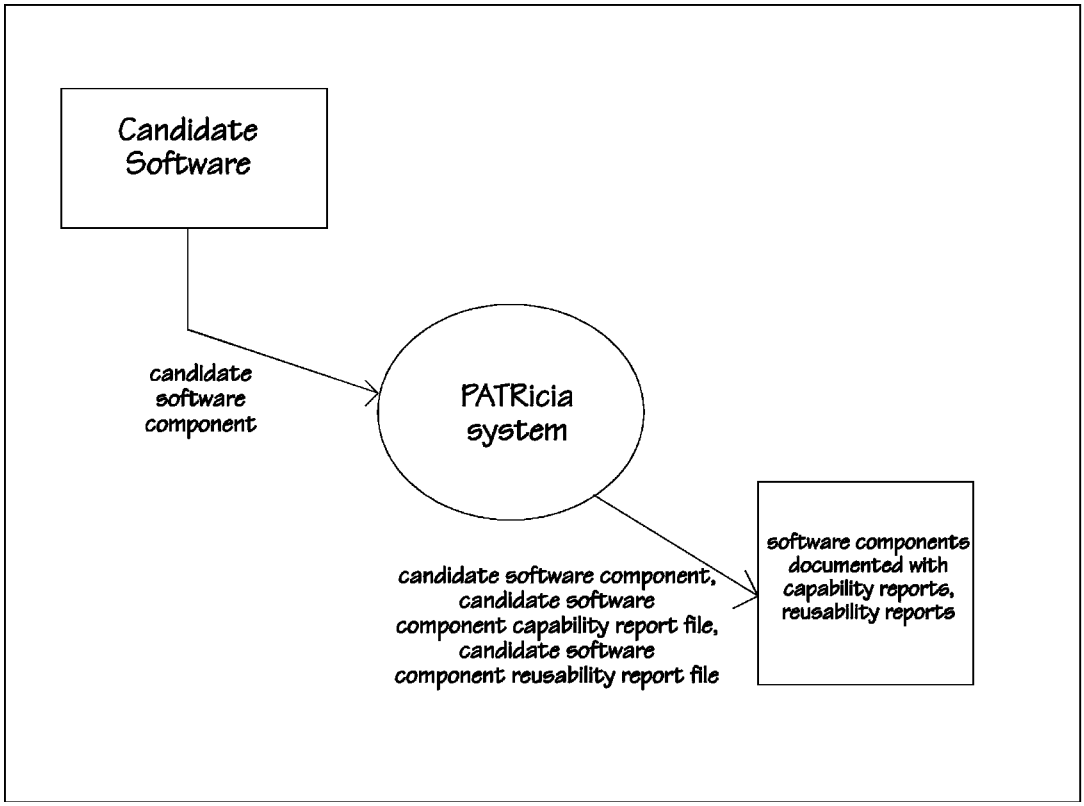


Figure 29. PATRicia System Context Diagram

The knowledge-base or domain base of the **PATR**icia system is a semantic network which is implemented in CLIPS version 6.0 objects and messages. The implementation of this knowledge-base is further described in section 6.1.4. Prior to the analysis of candidate software components, a domain base for the domain area of interest must be established. This is a labor-intensive task, that requires the identification and definition of important concepts related to the domain, the identification of links between concepts in order to form conceptual graphs that represent the interrelationship of knowledge within the domain, and the definition and weighting of inference links between concepts and conceptual graphs. An area of future research is the development of tools that would aid in the establishment of a domain base.

The **CHRiS** tool determines what functions a component is capable of performing, and the **Metrics Analyzer** tool provides metric data that will allow a knowledgeable software developer to

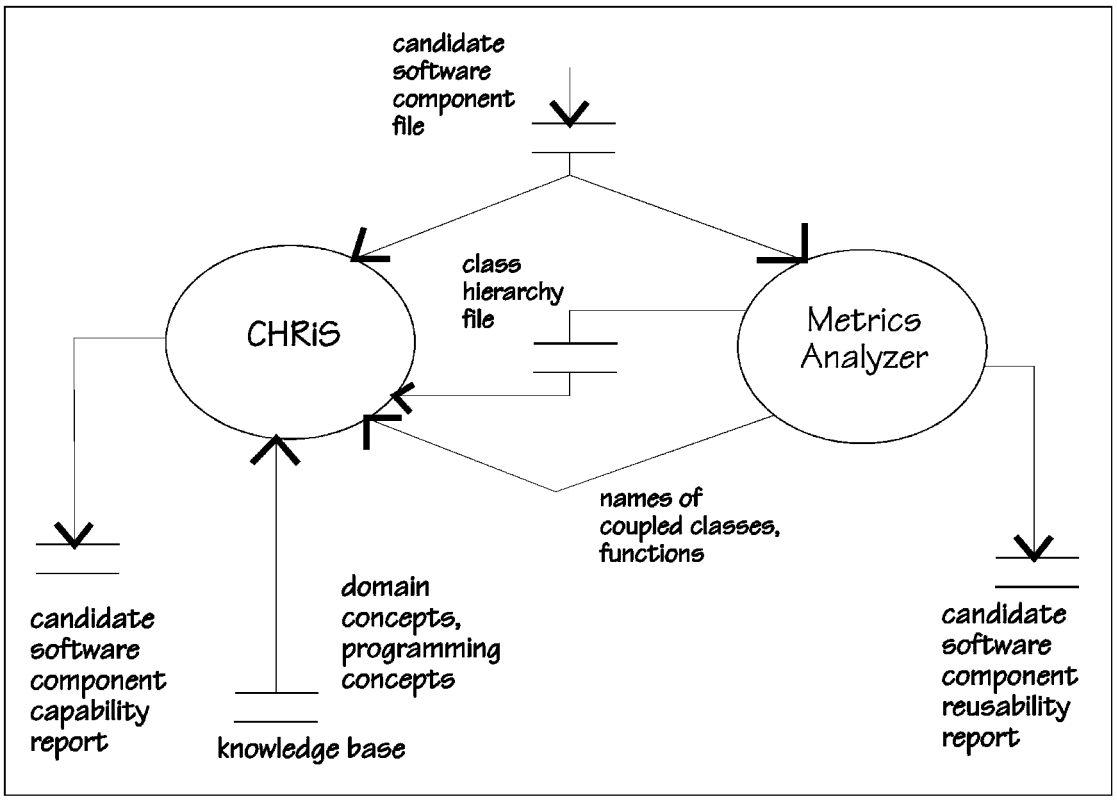


Figure 30. PATRicia System Level 1 Data Flow Diagram

determine if a software component is easily reusable. The operation of the **CHRiS** tool requires the prior establishment of an appropriate domain base.

The **CHRiS** tool provides several different reports for each candidate software component. One of the reports is a natural language description of the functions that a class can perform. Another report is a list of concepts that is covered by a class. These concepts could be used to help classify the component for insertion into a software reuse library. These reports are described further in section 6.1.5. The **Metrics Analyzer** tool provides three different reusability reports. One report is a list of the values of various object-oriented metrics for a particular class or class hierarchy. A second report contains the values determined for the reusability quality metrics for the current class, and a third report lists reusability quality metrics for the current class hierarchy. These reports are described further in section 6.2.4.

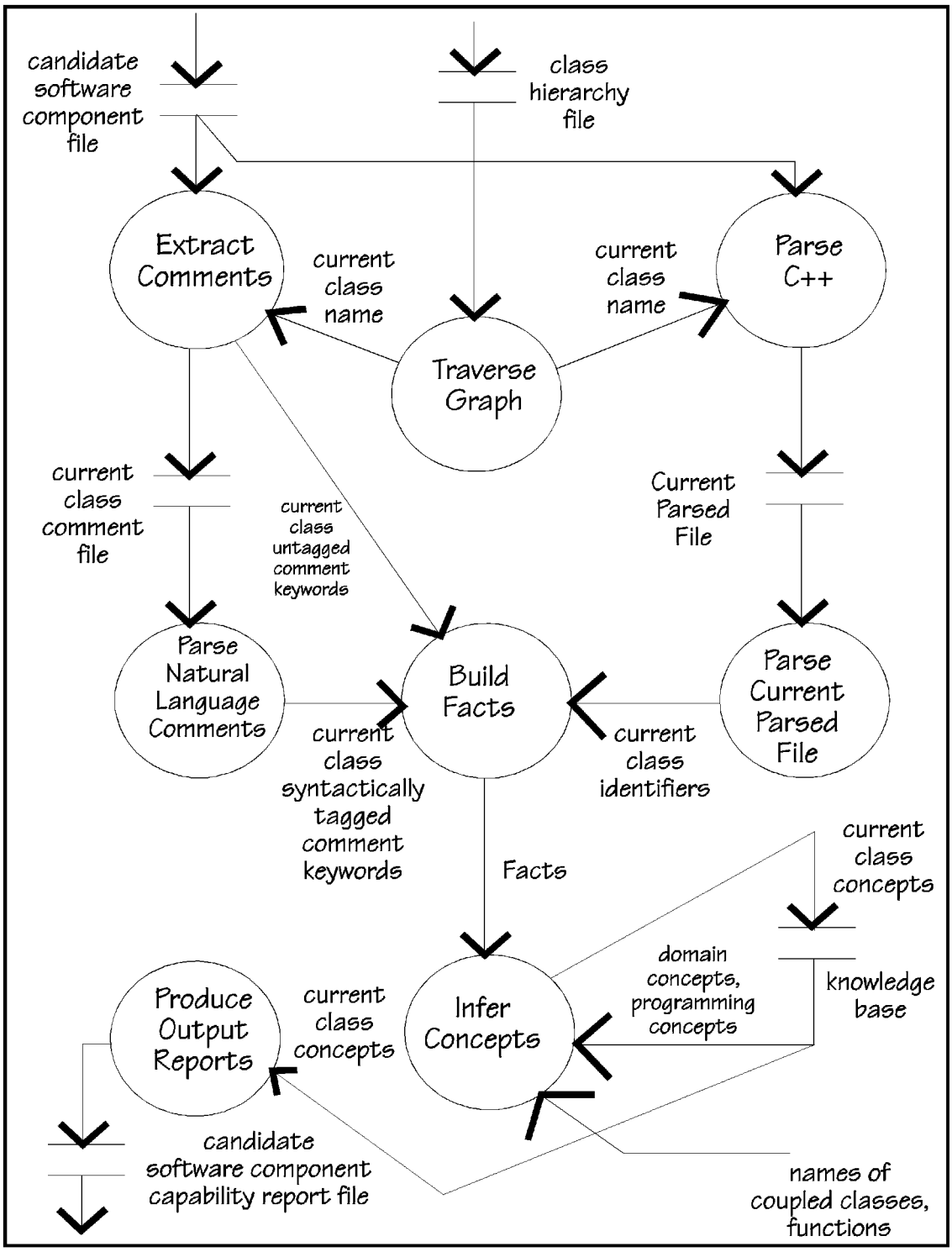


Figure 31. PATRicia System Level 2 Data Flow Diagram -- the CHRiS Tool

6.1. The Conceptual Hierarchy for Reuse including Semantics (CHRiS) Tool

6.1.1. Introduction

The operation of the **CHRiS** tool is shown in the level 2 data flow diagram in Figure 31.

The Parse C++ function shown in Figure 31 is implemented using the Brown University C++ parser, which produces output in the form of an abstract syntax tree, in a file in ASCII format. The Parse natural language function is implemented using the Sleator and Temperley natural language parser [101], which is a link grammar parser. A discussion of the selection and use of the Sleator and Temperley parser is in section 6.1.2 below. The domain base and the Infer concepts function are implemented in the CLIPS version 6.0 expert system shell.

The transforms on the data flow diagram in Figure 31 are described below.

6.1.1.1 Traverse Graph

Input to the Traverse graph transform is a class hierarchy chart in a file (provided by the Metrics Analyzer) for the C++ package being analyzed. The Traverse graph transform traverses the class hierarchy chart beginning at the base classes. All base classes are visited first, followed by immediately derived classes, and so on, using a breadth-first search. As each class in the hierarchy is visited, it is selected to be analyzed. The current class name is then output to the Parse C++ transform, and to the Extract Comments transform.

6.1.1.2. Parse C++

Inputs to the Parse C++ transform are the current candidate component file (a file containing C++ code), and the name of the current class to be analyzed. The Parse C++ transform extracts the current class from the candidate component, and uses the Brown University C++ parser to

produce an abstract syntax tree, in an ascii file, for that class. This abstract syntax tree file, or current parsed file, is then output to the Parse Current Parsed File transform.

6.1.1.3 Parse Current Parsed File

Input to the Parse Current Parsed File transform is the current parsed file, an ascii file that contains the abstract syntax tree of the current class. The Parse abstract syntax tree transform then extracts member function identifiers and attribute identifiers from the abstract syntax tree. Syntactical tagging of the member function identifiers is then performed, where possible. Output from the Parse abstract syntax tree transform is syntactically tagged (where possible) member function identifiers and attribute identifiers.

6.1.1.4 Extract Comments

Inputs to the Extract comments transform are the current candidate component file, and the name of the current class to be analyzed. Each sentence-form comment from the current class is selected and placed in a separate file, the current class comment file, which is output to the Parse natural language transform. Keywords from non-sentence-form comments are output directly to the Build facts transform.

6.1.1.5 Parse natural language

The Parse natural language transform uses the Sleator and Temperley natural language parser to parse comment sentences. Input to the Parse natural language transform are comments, in the current class comment file. Output from the Parse natural language transform are the parses of the comments, in the form of keywords from the comments, with each keyword tagged with sentence usage and syntactical information.

6.1.1.6 Build facts

Inputs to the Build facts transform are program keywords in the form of syntactically-tagged identifiers, keywords from sentence-form comments with usage and syntactical information, and keywords from non-sentence-form comments, that have no syntactical information. Keywords are asserted as facts in the CLIPS expert system shell. Examples of facts that contain syntactical information are as follows:

(term_part_of_speech (id_name user) (type noun) (usage subject) (location Application))

(term_part_of_speech (id_name accepts) (type verb) (usage) (location Application))

Keywords for which no syntactical information is available are also asserted as facts in the CLIPS expert system shell. An example of a fact that contains no syntactical information is as follows:

(term_identify (id_name application) (location Application))

6.1.1.7 Infer Concepts

Input to the Infer concepts transform are CLIPS facts, coupling information from the Metrics Analyzer (information as to the names of friend classes and functions), and programming concepts and domain concepts from the **PATR**icia system knowledge-base, which is implemented in CLIPS version 6.0 objects and messages. CLIPS rules present the facts in an efficient manner to the interface layer of the knowledge-base. Inferencing then proceeds through the knowledge-base. The inferencing scheme of the **PATR**icia system knowledge-base is described in section 6.1.4.

6.1.1.8 Produce Output Reports

Input to the Produce Output Reports transform are the current identified class concepts from the knowledge-base. Output from this transform are the **CHRiS** candidate software component capability reports.

6.1.2. Sleator and Temperley Natural Language Parser

6.1.2.1. Description of Operation

After an examination of natural language parsers, the Sleator and Temperley parser was chosen for use in the **CHRiS** tool for several reasons. The primary reason is that its dictionary is easily modified for sublanguage use. Second, it is a widely accepted and used parser, and third, it is a fast parser, on the order of $\mathbf{K*n^3}$, where **n** is the length of the sentence, and **K** is a constant dependent on the algorithm.

The Sleator and Temperley parser is a word-based, link grammar parser [101]. A link grammar has a power similar to that of a context-free grammar, and therefore the Sleator and Temperley parser is as powerful as most modern natural language parsers. Most comment-style sentences can be parsed with the Sleator and Temperley parser, as well as a large variety of more complicated sentences. The Sleator and Temperley parser handles: noun-verb agreement, questions, imperatives, complex and irregular verbs, many types of nouns, past-or-present participles in noun phrases, commas, a variety of adjective types, prepositions, adverbs, relative clauses, possessives, coordinating conjunctions, and more [101]. Figure 32 gives an example of the output of the Sleator and Temperley parser for an example comment sentence. In the example parse shown, "this" is a determiner that modifies "routine." The noun "routine" is the subject of the verb "reads." The noun "data" is the object of the verb "reads" and the determiner "the" modifies "data." The graphical output of the parser is also augmented by a non-graphical output which is used by the **CHRiS** tool.

Similarly to other natural language parsers, the Sleator and Temperley parser has a few limitations when parsing comments. For example, errors in punctuation can cause difficulties. Consider the following sentence, which is handled by the Sleator and Temperley parser:

It opens the file; then it reads the data a character at a time.

```
> Accepted (2 linkages, 2 with no P.P. violations)
  Linkage 1, cost vector = (0, 0, 2)

              +-----O----+
  +---D--+---S---+    +--D--+
  |    |       |   |    |
this routine.n reads the data.n

      /////         CLg     <---CLg--->  CL        routine.n
 (g)  this          D*u     <---D*u--->  D*u       routine.n
 (g)  routine.n     Ss      <---Ss---->  Ss        reads
 (g)  reads         O       <---Os---->  Os        data.n
 (g)  the           D       <---Dmu--->  Dmu       data.n
```

Figure 32. Parse of an Example Comment Sentence using the Sleator and Temperley Parser

If the semicolon between "file" and "then" is omitted, or replaced by a comma, then the Sleator and Temperley parser rejects the sentence.

Yet another difficulty with the Sleator and Temperley parser is that it doesn't handle parentheses. It treats the words within a pair of parentheses as a single word, and gives a "not in the dictionary" error message.

One particular problem with the parsing of comment sentences is their telegraphic quality, e.g., connector words such as "the" are often left out. This tends to confuse the Sleator and Temperley parser. For example, the Sleator and Temperley parser does not accept the following comments:

1) Scan argument declaration for template.

2) Scan the argument declaration for template.

3) Scan argument declaration for the template

However, the Sleator and Temperley parser does accept:

4) Scan the argument declaration for the template.

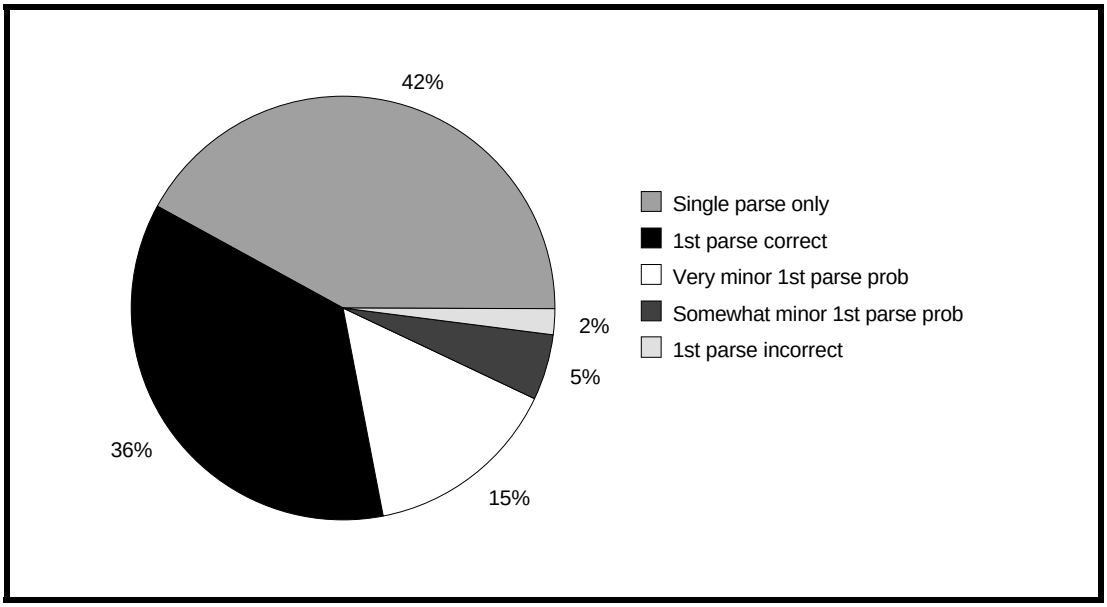


Figure 33. Results of the Sleator and Temperley Parser on Parsable Comments

The **CHRiS** tool has been designed to anticipate some of these difficulties, and puts the sentence in a structure more likely to be accepted by the parser.

6.1.2.2. Study of the Application of the Sleator and Temperley Parser to Comments

Prior to the selection of the Sleator and Temperley parser for use in the **PATR**icia system, a study applying the parser to a set of typical comments was performed. The purpose of the study was to determine how well the parser worked on parsable comments (as opposed to comments that were rejected by the parser).

This study examined the first 20 parsable comments taken from each of five different sources: three graphical user interface packages [8][102][110], a parallelization tool [88], and comments from a C++ textbook [112]. In all, 83 parsable comments were examined. The overall results of this study are shown in Figure 33.

For 78% of comments, the first parse was the correct parse. For 93% of comments, the first parse provides useful information, with very minor errors from the standpoint of the **CHRiS** tool.

These errors include such errors as a prepositional phrase that is attached to the wrong word, an adverb that modifies the wrong verb, or an adjective treated as part of a noun phrase (that is, as part of the noun). This is not important since the **CHRiS** tool does not currently look at the prepositional phrase attachment provided by the parser, but rather, prepositional phrase attachment is handled semantically within the inferencing engine of the knowledge-base. There is very little adverbial usage in the knowledge-base of **CHRiS** since adverbs qualify verbs, and this is also generally not needed for a knowledge-base whose purpose is examining computer software. The treatment of an adjective as part of a noun phrase labels the adjective as an adjectival, which is a noun performing as an adjective. The **CHRiS** tool simply treats adjectivals the same as adjectives.

For 5% of the parsable comments, some very minor erroneous information would be presented to the **CHRiS** tool; however, most of the parsing information even in this 5% segment would still be correct. Only for 2% of parsable comment sentences does the Sleator and Temperley parser truly fail from the standpoint of the **CHRiS** tool.

The dictionary used with the Sleator and Temperley parser for this study was the dictionary that was supplied with the parser, with the addition of a few words. In actual application, the parses would be even more correct with a sublanguage restriction on the dictionary.

6.1.3. Syntactic Phase--Description of Operation

6.1.3.1. Order of Understanding

The **CHRiS** tool uses the class hierarchy as the focus for understanding, starting with base classes first, then derived classes. This results in an almost top-down understanding process, since high level concepts are understood first, followed by lower level concepts. This has two advantages. In the first place, the set of concepts that makes up the class is limited. No concept associated with another class that is not part of this class' hierarchy (unless a friend class, which is also defined within the current class) could possibly be part of the makeup of the class. Also,

after the concepts related to a particular class have been identified, a shrinking of the domain can occur when attempting to understand derived classes, which can lead to a quicker and more accurate understanding of the derived classes. This domain shrinking has been investigated and is described in the knowledge-base discussion in section 6.1.4 below.

6.1.3.2. Comment Heuristics

Comments are extracted from the current class definition and its member function definitions by the Extract comments and keywords function. Various heuristics are applied to each comment and the keywords within the comments in order to make them more acceptable to the Sleator and Temperley parser. An example of a definition style of comment is shown below:

Read_Data: Reads the data.

or

Read_Data--Reads the data

Neither of these comments is acceptable as written to the natural language parser. Both of the above are handled by having the sentence replaced by the following sentence:

Read_Data reads the data.

The software that implements this heuristic looks to see if a colon or dash is found after the first word in a sentence, then removes the colon or dash, and converts the first character in the following word to lower case. The Read_Data identifier itself will be understood separately, as part of the identifier understanding process.

Another heuristic relates to sentences of two words. First the natural language parser is applied to the sentence. If the sentence is rejected, then the determiner "the" is inserted between the two words within the sentence, and the natural language parser is again applied. This is identical to a heuristic that is used in the identifier understanding process. One other heuristic employed is placing the word "This" in sentences that begin with a plural verb, and uncapitalizing the verb ("Read the data." thus becomes "This reads the data.") This is determined in a very simple manner, e.g., if the original sentence is rejected by the natural language parser, then the

software looks at the sentence to see if the first word ends with an 's'. If this is so, the word "This" is added to the beginning of the sentence, and the natural language parser is again applied to the sentence.

Other heuristics involve the removal of parenthetical information from a comment before the application of the natural language parser, and the appending of each comment sentence with a period character before the application of the natural language parser. Many comments in computer software do not have a sentence terminated with a period. Also, many comment sentences do not begin with a capital letter, and ensuring that they do is another heuristic.

This leads to another comment processing difficulty, which is the difference between comments that occur on a single line and comments that continue between several lines. Consider a C style comment of the following type:

```
/* The following routine extracts sentences from the header buffer
   and sends them to the natural language parser.  If parsable,
   additional information about the usage of keywords is
   included in the facts sent to CLIPS.
   Certain types of sentence based heuristics are handled here.  */
```

In this case, where a sentence is extended over multiple lines, each sentence must be extracted separately (a sentence here is a sequence of characters terminated by a period), and sent separately to the natural language parser.

A harder challenge comes from C++ style comments that extended across multiple lines. For example,

```
// We begin the
// sentence on one line and
// complete it on another.  Worse yet,
// we have two sentences in the
// same manner.
```

This type of sentence is handled by a preprocessor that is applied to the software prior to the **CHRiS** tool being run on the software. C++ style comments that appear sequentially, on lines

that did not include actual code, have the C++ comment characters removed, and a C style comment character added at the beginning and the end of the sequence of comments. They can then be processed by **CHRiS** in the same manner as the C style comments discussed above.

6.1.3.3. Identifier Heuristics

Class identifiers are determined by use of the Brown University C++ parser. The C++ parser is applied to the candidate component, and produces an abstract syntax tree in an ASCII file. The abstract syntax tree file is then parsed to retrieve the identifiers of the current class, primarily the member function and attribute identifiers.

There are many heuristics related to the processing of identifiers. First, the identifiers (as well as keywords extracted from comments) are broken into separate keywords based on underscore and capitalization information. For example, "read_data" would be broken into the two keywords "read" and "data," with the break occurring at the underscore. ReadData would be broken into the two keywords "read" and "data" based on capitalization information. At this point, for member function identifiers, the algorithm described in Chapter IV, Figure 21, is applied.

6.1.3.4. CLIPS Facts

Parsing information which is derived from the natural language parser is then built into facts which are presented to the CLIPS expert system shell. The facts include the parsing information such as the part of speech (noun, adjective, verb, etc.) and sentence usage information (subject, object, etc.), as well as location information (the location is normally the name of the class that is presently being understood). Some example facts are:

(term_identify (id_name application) (location Application))
(term_part_of_speech (id_name user) (type noun) (usage subject) (location Application))
(term_part_of_speech (id_name accepts) (type verb) (usage) (location Application))

CLIPS rules present the information in an efficient manner to the interface layer of the knowledge-base. Inferencing then proceeds through the knowledge-base.

6.1.4. Semantic Processing--Knowledge-Base

The **CHRiS** knowledge-base is itself object-oriented, and is implemented in CLIPS version 6.0 objects. The inferencing that is required for semantic processing is implemented as spreading activation within the knowledge-base. The spreading activation is implemented within the knowledge-base as messages sent from one CLIPS object to another.

6.1.4.1. Interface Layer

The **CHRiS** knowledge-base is a weighted, hierarchical semantic network. The semantic network has an interface layer of syntactically-tagged keywords. The CLIPS class definition for a keyword node in the interface layer is shown in Figure 34.

A description of the various portions of a keyword interface node starts with the id_name, type, and usage slots which together categorize a particular keyword interface node. For example, the keyword node "window" can be a noun (type) and can be used as a subject, object, or object of a preposition (usage). Usage is a multislot since the various usage possibilities often are not used to break a particular keyword interface node into separate keyword interface nodes. In CLIPS, a "multislot" can contain multiple values, whereas a "slot" can contain only one value. This is a definition of a CLIPS version 6.0 object; however, the "slot" notation traces to Artificial Intelligence frames, which are very similar to objects. "Type" is a multislot since in some cases it is not necessary to differentiate between different parts of speech (such as noun versus adjective),

```
(defclass TERM_NODE (is-a USER)

        (role concrete)
        (pattern-match reactive)

        (slot id-name (create-accessor read-write) ) ; Keyword string

        ;  The certainty of a concept being matched is increased if
        ;   the keywords corresponding to that concept are found multiple
        ;   times in the current area of interest in the code under examination
        (slot number_times_found (type NUMBER)
                                 (create-accessor read-write)
                                 (default 0) )
        (slot overall_number_times_found (type NUMBER)
                                         (create-accessor read-write)
                                         (default 0) )

        (multislot type (create-accessor read-write) )
        (multislot usage (create-accessor read-write) )
        (slot match (create-accessor read-write) (default FALSE) )
        (multislot location (create-accessor read-write) )
        (multislot restriction (create-accessor read-write) )
        (slot binding (create-accessor read-write) )
        (slot level (type NUMBER) (create-accessor read-write) (default 0) )
        (slot printed (create-accessor read-write) (default "FALSE") )
        (multislot up-links (create-accessor read-write) )
        (slot module_defined_in (create-accessor read-write) (default A) )

) ; end class TERM_NODE
```

Figure 34. CLIPS Class Definition for a Keyword Interface Node in the Interface Layer of the CHRiS Semantic Net

```
(defclass NODE (is-a USER)
        (role concrete)
        (pattern-match reactive)
        (slot id_name (create-accessor read-write)
                        (default NIL) )        ; Concept's generic name
        (slot threshold (type NUMBER) (create-accessor read-write)
                                (default 1.0) )
        (slot sum_of_input_weights (type NUMBER) (create-accessor read-write)
                                (default 0.0) )
        (slot sum_of_input_weights_due_to_NODE_inference
                                (type NUMBER) (create-accessor read-write)
                                (default 0.0) )
        (slot match (create-accessor read-write)
                (default FALSE) )
        (multislot location (create-accessor read-write) )
        (multislot restriction (create-accessor read-write) )
         (slot level (type NUMBER) (create-accessor read-write) (default 0) )
        (slot major_or_minor (create-accessor read-write) (default minor) )
        (slot module_defined_in (create-accessor read-write) (default A) )
         (multislot facts (create-accessor read-write) )
        (slot printed (create-accessor read-write) (default "FALSE") )
         (slot nl_generation (create-accessor read-write )
        (multislot up_links (create-accessor read-write) )
        (multislot down_links (create-accessor read-write) )
) ; end class NODE
```

Figure 35. CLIPS Class Definition for an Interior Concept Node in the CHRiS Semantic Net

although most commonly these parts of speech will differentiate similar id_names into two different keyword interface nodes.

The number_times_found and overall_number_times_found slots are counts of the number of times a particular keyword interface node is fired. The number_times_found is the number of times a particular keyword, with associated usage and part of speech information, is found in the fact base for the current CLIPS "run" or inference session. The overall_number_times_found is

the count of the number of times a particular keyword, with associated usage and part of speech information, is found in the fact base after several CLIPS "runs." The number_times_found slot is used to increment the overall_number_times_found slot. The number_times_found slot is cleared after each CLIPS run. The certainty of a concept being matched is increased if the keywords corresponding to that concept are found multiple times in the current area of interest within the software being examined.

The match slot is set to TRUE when a fact in the current fact base contains a keyword that matches the current id_name, type, and usage. The match slot is cleared after every matching set.

The printed slot is used during the generation of a CLIPS keyword report. It is possible for a keyword interface node to be visited multiple times during the keyword report generation, and the printed slot prevents multiple reports of the same node.

6.1.4.2. Domain Shrinking

The binding, restriction, and module_defined_in slots are used when the knowledge-base is restricted, based on earlier concepts identified. Restriction based on previous concepts identified is possible, but difficult in the current implementation of the knowledge-base.

The identification of certain concepts would tend to indicate that certain other concepts are more likely to occur, and that certain other concepts are less likely to occur. What would be desirable would be to effectively shrink the knowledge-base after certain concepts were identified by heightening the sensitivity of certain other concepts, which is a schemata-like approach. Heightening the sensitivity of certain concepts would speed the identification of the more likely concepts, then the less likely concepts could be searched only if the more highly sensitive concepts were not matched.

In CLIPS rules, the heightening or reducing of the sensitivity of a rule is called the "salience" of the rule [28][29][30][31]. In CLIPS, the agenda is the list of all rules which have had their conditions satisfied and have not yet been executed. The agenda acts similarly to a stack, i.e. the top rule on the agenda is the first one to be executed. When a rule is newly activated, it is placed

on the agenda. Newly activated rules are placed above all rules of lower salience, and below all rules of higher salience. By use of the SetSalienceEvaluation command (set to when-activated), dynamic salience of rules can be achieved. This allows a re-evaluation of a rule's salience during processing, and thus certain rules can acquire priority over other rules. Conditions could be set during the firing of a higher priority rule that would then prevent rules with lower salience from firing.

However, the **CHRiS** semantic net was implemented in CLIPS version 6.0 objects and messages instead of CLIPS rules, for several reasons. First, it seemed intuitive to compare object-oriented software versus an object-oriented knowledge-base. A concept in one environment tends to map naturally to a concept in the other environment. Second, the use of objects allowed a simple implementation of the inferencing required via messages between objects. Third, the slots of the objects allowed for easy matching of concepts with several criteria. If the semantic net was implemented in a rule-based manner, each of these objects would have required several separate rules to implement. Unfortunately, an object in CLIPS version 6.0 can not receive higher or lower salience. This precludes an implementation of the restriction of a knowledge-base similar to that described above.

A more restrictive version of knowledge-base restriction is possible by the use of CLIPS modules [30]. CLIPS modules allow a knowledge-base to be partitioned. They allow a set of constructs to be grouped together such that explicit control can be maintained over restricting the access of the constructs by other modules. The control provided by a CLIPS module is similar to scoping in C or Pascal. However, with the CLIPS modules scoping is hierarchical, and operates in one direction only, i.e., if module A can see module B's constructs, then module B cannot see module A's constructs. Unless specifically exported and imported, the constructs of one module may not be used by another module. Modules can be used to provide execution control. However, since a module once defined cannot be deleted or redefined, all knowledge-base shrinking must be planned for at the time of the knowledge-base design, and is very restrictive. Constructs that are imported into one module must have been defined for export in the module

exporting them at the time that module is defined. Thus dynamic exportation from one module to another is quite difficult.

As part of this research, a non-dynamic knowledge-base shrinking mechanism was implemented and tested. This domain shrinking information is defined statically within the knowledge-base. Interior concepts, those that are not part of the interface layer, are divided into two categories, major and minor. Within the context of the **CHRiS** knowledge-base, a major concept is a concept that can be used to shrink a domain. The domain shrinking is performed by the use of CLIPS modules. CLIPS modules must be defined *a priori*, with all necessary constructs that the module uses defined when the module is defined. Each major interior concept corresponds to a CLIPS module. Only a few concepts within a particular knowledge-base should be defined as major concepts, otherwise an exponential increase in the number of CLIPS modules occurs.

However, this method is very restrictive, since it does not allow lower level concepts (those within other modules) to fire if the higher level concepts are not found in the current module. This means that there must be a strong certainty that the current knowledge-base restriction (module selection) is appropriate.

Domain shrinking of another kind has been implemented by the use of downlinks. Each major concept can have associated downlinks to all of its main sub-concepts (and those sub-concepts can have downlinks to their sub-concepts). These downlinks are used to send restriction information to sub-concepts. For example, if a particular major concept has been identified, the name of that major concept can be sent to its subconcepts. Then only concepts containing the name of that major concept in its restriction information will be tagged as identified during future inferencing. This is not as effective as the use of CLIPS modules. The use of restriction information can reduce false identification of concepts not belonging to the new sub-domain; however, the concepts that do not belong to the new sub-domain are actually examined by the CLIPS inferencing engine, as well as the concepts that do belong to the new sub-domain. Thus no actual comparison time is saved. The only way to prevent the CLIPS inferencing engine from examining concepts (defined as an instance of a CLIPS class) not in the sub-domain is by the use of a CLIPS

```
(defclass       CONCEPTUAL_RELATION (is-a USER)

                (role concrete)
                (pattern-match reactive)

                ; Relations between concepts
                (slot relation_type (create-accessor read-write)
                                    (visibility public)
                                    (default NIL) )

                (slot cg_printed (create-accessor read-write)
                                 (default NONE) )

                (multislot concept_number (create-accessor read-write)
                                          (visibility public) )

                (slot from  (create-accessor read-write)
                            (visibility public) )
                (slot to    (create-accessor read-write)
                            (visibility public) )
) ; end class CONCEPTUAL_RELATION
```

Figure 36. CLIPS Class Definition for a Conceptual Relation in the CHRiS Semantic Net

module defined "around" the sub-domain, in other words including all concepts in that sub-domain, but not in other domains or sub-domains.

6.1.4.3. Semantic Net Implementation

Three kinds of links are used within the semantic net: "uplinks," or inferencing links, which are used to form super-concepts out of sub-concepts, "downlinks," which are used in domain shrinking and for report printing, and "conceptual relations," which link concepts in a conceptual

graph. Each of these links, as well as the keyword interface nodes, and interior concept nodes, is a CLIPS version 6.0 object. See Figure 35 for the CLIPS class definition associated with with an interior concept node. Note that the interior concept node class serves as a template for concepts within a conceptual graph.

Each uplink has a weight associated with it. The weights on the uplinks of the semantic net, together with the threshold values in concept nodes, are used as the main inferencing mechanism of the semantic net. Messages associated with CLIPS classes are used to implement this inferencing mechanism, which is a form of spreading activation. See Chapter IV for a description of spreading activation within the semantic net.

The slots sum_of_input_weights and sum_of_input_weights_due_to_NODE_inference are related to spreading activation within the semantic net. The sum_of_input_weights is compared to the threshold value in order to determine whether or not a particular interior concept node should be asserted. The sum_of_input_weights_due_to_NODE_inference contains the weight based on uplinks from other interior concept nodes, whereas the sum_of_input_weights contains the weight based on all uplinks, and includes those from the keyword interface layer as well as from interior concept nodes. At the end of a run the sum_of_input_weights is set to the value from the sum_of_input_weights_due_to_NODE_inferencing. The idea here is that keywords in the interface layer are valid only for a single "run," whereas recognized concepts are valid for multiple "runs."

Each interior concept node within the semantic net has associated English text defining the operation of that concept within the current domain, e.g., see the multislot facts in Figure 35. This text gives the high-level definition of the concept, without as much low-level detail as is found in the software. The English text associated with each identified concept is included in a final report generated by **CHRiS**.

Both interior concept nodes and keyword interface nodes allow for information regarding the location where a node was specified. Currently this operates only down to the class level, although hooks are included for the addition of finer-grain (within a class) information. Location

is identified by class name. If finer-grain location is later added, it will probably also be necessary to add information as to the type of location, i.e., class member function, class data definition, etc.

Figure 36 contains the CLIPS class definition for a conceptual relation. The relation_type slot contains the type of conceptual relation. The conceptual relations employed in the conceptual graphs within the semantic network include AGNT (for agent), OBJ (for object), ATTR (for attribute), LOC (for location), IS-A (this is a type relation), and IS_PART_OF (this is a whole-part relation). See Chapter IV for the use of IS-A and IS_PART_OF within the **CHRiS** inferencing engine.

The multislot concept_number is used to differentiate between different conceptual graphs. A conceptual relation may be a member of multiple conceptual graphs, and the distinction between different conceptual graphs is made statically at the time of knowledge-base design.

The "from" slot and the "to" multislot list the different concepts linked by the conceptual relation. In some situations it was convenient for the same conceptual relation type to connect from one concept to multiple concepts; for example, the situation where a concept has multiple agents or objects.

6.1.4.4. Semantic Net Design for Selected Domains

In order to take full advantage of the word usage information derived from the natural language parser, a careful design of the semantic net for a particular domain is required. The format of possible comments must be considered during the knowledge-base design, particularly when considering uplinks from the interface layer to interior concepts. This can be a fairly difficult task, since it is necessary to consider not only possible comments that might occur in the current domain, but also possible comments that might occur in other domains that would tend to cause a false keyword identification in the current domain.

For example, consider a listen mode menu concept, and assume that a semantic net is being designed to handle the domain of Graphical User Interfaces (of which multimedia graphical user

Class wxObject

1) object, with weight 1.000000, at location(s) wxObject found
Facts are:
"GUI item that can be placed, hid, redrawn, scaled, etc."

Class wxbWindow

1) activate, with weight 1.000000 at location(s) wxbWindow found

2) box_frame, with weight 1.000000 at location(s) wxbWindow found
Facts are:
"The part of a dialog box that defines the box's boundaries and separates each box from the other boxes and windows that it may overlap. Note the modeless dialog boxes have complete frames with title bars; modal boxes often have thin borders and no title bars"

3) control_menu, with weight 1.500000, at location(s) wxbWindow found
Facts are:
"A menu that controls whether the current window is opened or closed. Some GUIs have a close button instead of a control menu"

. . .

Class wxWindow

. . .

9) cursor, with weight 1.000000, at location(s) wxbWindow wxWindow found
Facts are:
"Marks a location on a screen or in a window"

10) dialog_box,with weight 9.200000,at location(s) wxbWindow wxWindow found
Facts are:
"A window in which the computer can present several alternatives, ask for more information, or warn the user an error has occurred.

Figure 37. Portions of a CHRiS Concept Report

interfaces are a sub-domain). In order to take advantage of word usage information, the knowledge-base designer might consider (for example) the following comments:

1) This is a listen mode menu.

2) This menu has a listen mode.

3) Listen to the port in the basic mode to read the menu.

The first comment is definitely related to the domain in question. The second comment might be related to the domain concept (depending on context there are other interpretations). The third comment is a comment that would be found in some type of communications interface software, and has no direction to the GUI domain. In the first sentence, the Sleator and Temperley parser, with the appropriate dictionary, recognizes "listen" and "mode" as adjectives, with "menu" as a noun with usage object. In the second sentence, the parser recognizes "menu" as a noun with usage subject, with "listen" as an adjective, and "mode" as a noun with usage object. In the third sentence, the parser recognizes "listen" as a verb, "mode" as a noun with usage object of preposition, and "menu" as a noun with usage object. A knowledge-base that differentiated the listen mode menu concept by requiring a keyword interface node "listen" with usage adjective would assure that the third comment did not result in a false identification of the listen mode menu concept. By the use of simple English syntax information, such as that provided by the Sleator and Temperley parser, it is not possible to determine whether the second comment is related to the GUI domain or not, unless other contextual information is also available. Thus the understanding of the second sentence is a more semantically-related problem.

6.1.5. Reports Produced by CHRiS

CHRiS produces four reports. The first report is a list of standard concepts identified for each class, that could be used to categorize the class for insertion into a class library. The second report is a description of the functionality of the class, in natural language. The third report is a list of all keywords from the interface layer that have been identified as associated with the class. The fourth report shows the links between nodes in the semantic net that led to certain concepts

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 38. CHRiS Description of Functionality Report -- Natural Language Generation

being identified. This information can serve 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.

6.1.5.1. Concept Report

Figure 37 shows an example of various portions of a concept report. This report is a list of standard concepts that have been identified as belonging to a particular class, along with English text descriptions of the concepts. The weight determined by the inferencing mechanism for each concept is also shown. The idea is that concepts with higher weights are more strongly identified.

The concept report includes reports for all classes that have been examined in the current software component under examination. If a concept has been found multiple times, in multiple

classes, then the concept report lists all classes in which the concept was found at the time of the current class report. This means that derived classes will have a locations list that will occasionally list a concept that appears both in an inherited class and in the local class. Also, concepts that were inherited from another class, but that were not explicitly discovered in the current, derived class, will be listed, but the locations list will include only the inherited class, and not the current class.

6.1.5.2. Natural Language Generation

Figure 38 provides an example of the natural language description of functionality report that is provided by **CHRiS**. This report lists and describes the concepts that are contained in a class, and it describes the functions that are performed by a class. Inherited concepts and functions are reported also in the report for a derived class. This report is generated using **CHRiS**' simple natural language generation facility.

CHRiS performs natural language generation very simply. Each sentence in the functionality report is related to a numbered conceptual graph in the **CHRiS** semantic net that has at least one concept identified.

The functionality report is generated in present tense only. Verbs (concepts in a conceptual graph that are pointed to by the "from" slot in an AGNT or OBJ conceptual relation) default to singular. In cases where an interior concept node has multiple AGNT conceptual relations, the plural of a verb is used (usually, the 's' form or the no 's' form of the verb is chosen, although occasionally a verb has specified (in the knowledge-base) the 'es' form. For example, "open" is plural, and "opens" is singular, thus the 's' form is the default). ATTR conceptual relations that begin at the verb are considered to point to adverbs. There is a nl_generation slot in the interior concept node class definition (see Figure 35). This contains specialized information about the form in which each concept is to be generated. For example, adverbs can be generated with "ly" appended, or without. For example, something can move "slowly," and it can move "up." Both are adverbs modifying "move," since they describe the move. However, one adverb has "ly"

appended and the other does not. Adverbs default to having "ly" appended. In the nl_generation slot, however, a "notly" entry is possible. This prevents move "up" from being generated as move "uply," for example.

Each noun concept (concepts in a conceptual graph that are pointed to by the "to" slot in an AGNT or OBJ conceptual relation) has a determiner generated. Usually, when the determiner is generated as part of a sentence that also contains a verb, then the determiner generated is indefinite, either 'a' or 'an', and the selection of 'a' versus 'an' is made based on whether the following concept begins with a consonant or a vowel. For example, such constructs as these would be generated, with appropriate determiners:

-- minimizes a window

or

-- activates an object

When a noun concept is determined as belonging to a class, and the verb in the conceptual graph is either not present, or has not been identified as belonging to the class, then the determiner generated is specific. The following is an example of such a construct:

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

In this case the concept "text" did not have any identified verbs in conceptual graphs appended to it. Thus it was generated as "the concept text," and the determiner selected was "the," which is specific.

However, "text" did have ATTR conceptual relations that connected to identified concepts. An ATTR conceptual relation from concept A to concept B means that concept B is a quality that can describe concept A. This construct is generated by the use of the phrase "that can be described by," then an appropriate indefinite determiner, followed by the word, followed by the word "descriptor."

Multiple OBJ conceptual relations result in a compound predicate. For example:

-- erases a line and a window.

For more than two OBJ conceptual relations, the *and* construct simply continues:

-- erases a box and a circle and a rectangle and a cylinder

When a numbered conceptual graph that has at least one concept identified also contains other concepts that were not identified, then those concepts are used to produce hints regarding how the class might operate. For example, the report for one class generates the following sentences:

-- moves up a mouse and an object

> HINT: It is possible to move a mouse that can own a button and an object that can be described by an orientation descriptor and a position descriptor and a size descriptor and a style descriptor.

In the conceptual graph that represents the following, the concepts "move," "up," "mouse," and "object" were identified. However, each noun concept had various ATTR links to concepts that were not themselves identified. This resulted in the given hint. Hints are specified in the form "can be" and "it is possible" to ensure that they are not confused with identified concepts.

The **CHRiS** natural language generation facility is not intended to produce general natural language; rather, it is intended to produce a simple report of the functionality of a class. It has sufficient power to enable it to produce such a report. Some improvements are possible; for example, the compound subject handling could be extended to use the comma construct rather than always the *and* construct. However, the natural language generation that has been provided is sufficient for the task of report generation.

6.1.5.3. Other CHRiS Reports

The **CHRiS** tool also produces two other reports: a list of all keywords from the interface layer that have been identified as associated with the class, and another report that shows the links between nodes in the semantic net that led to certain concepts being identified. The keyword report lists each keyword identified as having been associated with the class, and the number of times the keyword was found. The other report (typically called a "full" report) can serve 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.

6.1.6. Limitations

Most of the limitations of **CHRiS** are caused by the limitations of the various tools it uses. For example, the Sleator and Temperley parser has several limitations as to the sentences it will handle; the most common problem is that it does not handle telegraphic-style sentences well (see section 6.1.2 above for a greater discussion of the limitations of the Sleator and Temperley parser).

Another limitation is caused by the Brown University C++ parser, which is used by **CHRiS** primarily to extract identifiers from C++ software. The Brown University C++ parser handles most common C++ constructs; however, there are several that it does not handle. It has difficulty compiling some XWindows files. It will not handle DOS-style code constructs such as "far" pointers.

One tool that is not normally considered a part of **CHRiS**, but whose operation does impact **CHRiS** is the PCMETRIC™ tool [89]. The PCMETRIC™ tool is used by the **Metric Analyzer** to generate various low level object-oriented metrics. One such metric, the class hierarchy graph, is used by **CHRiS** to guide its understanding process. PCMETRIC™ seems to have an occasional problem with large files.

The major limitation of **CHRiS**, which is both its strength and its weakness, is the knowledge-base. The form of the knowledge-base, that of a semantic net consisting of conceptual graphs, seems quite powerful, and appropriate to the task of identifying reusable components. However, a large amount of work is required to develop the knowledge-base in order to achieve sufficient knowledge for even a fairly small domain. The development of a sufficient knowledge-base for the area of Graphical User Interfaces was a task that required well over three months work. Semantic, conceptual graph tools are needed to ease the knowledge-base development process.

CHRiS currently is a research prototype, that was intended as a proof of concept. Thus the user interface of **CHRiS** has not been carefully addressed. The user interface is currently a simple command line interface.

6.2. The Metrics Analyzer Tool

6.2.1. Introduction

The purpose of the **Metrics Analyzer** tool is to determine the extent to which object-oriented software components are reusable. The **Metrics Analyzer** tool implements Reusability-in-the-Class, Reusability-in-the-Original-System, and Reusability-in-the-Hierarchy. However, since the user interface to the **PATR**icia system is a very simple, command line interface, the **Metrics Analyzer** does not currently implement Reusability-in-the-Subsystem. The lack of an easily defined way to select a particular subsystem is the reason Reusability-in-the-Subsystem was not implemented. However, this should be a simple extension to Reusability-in-the-Hierarchy when a graphical interface is implemented for the **PATR**icia system.

The **Metrics Analyzer** uses a combination of commercial and shareware tools, new C and C++ software, and various lex-generated parsers. Figure 39 is a level 2 Data Flow Diagram of the **Metrics Analyzer** within the **PATR**icia system. The Parse C++ function shown in Figure 39 is implemented using the Brown University C++ parser, which produces output in the form of an abstract syntax tree, in a file in ASCII format. The Collect OO metrics function calculates several object-oriented metrics. For some low level object-oriented metrics, it uses the PCMETRIC™ tool [89], which is a commercial tool that provides various simple object-oriented metrics for C++.

6.2.1.1 Parse C++

Input to the Parse C++ transform is a C++ candidate software component. The Parse C++ transform is implemented using the Brown University C++ parser. Output from the Parse C++ transform is an abstract syntax tree of the candidate component, in an ascii file.

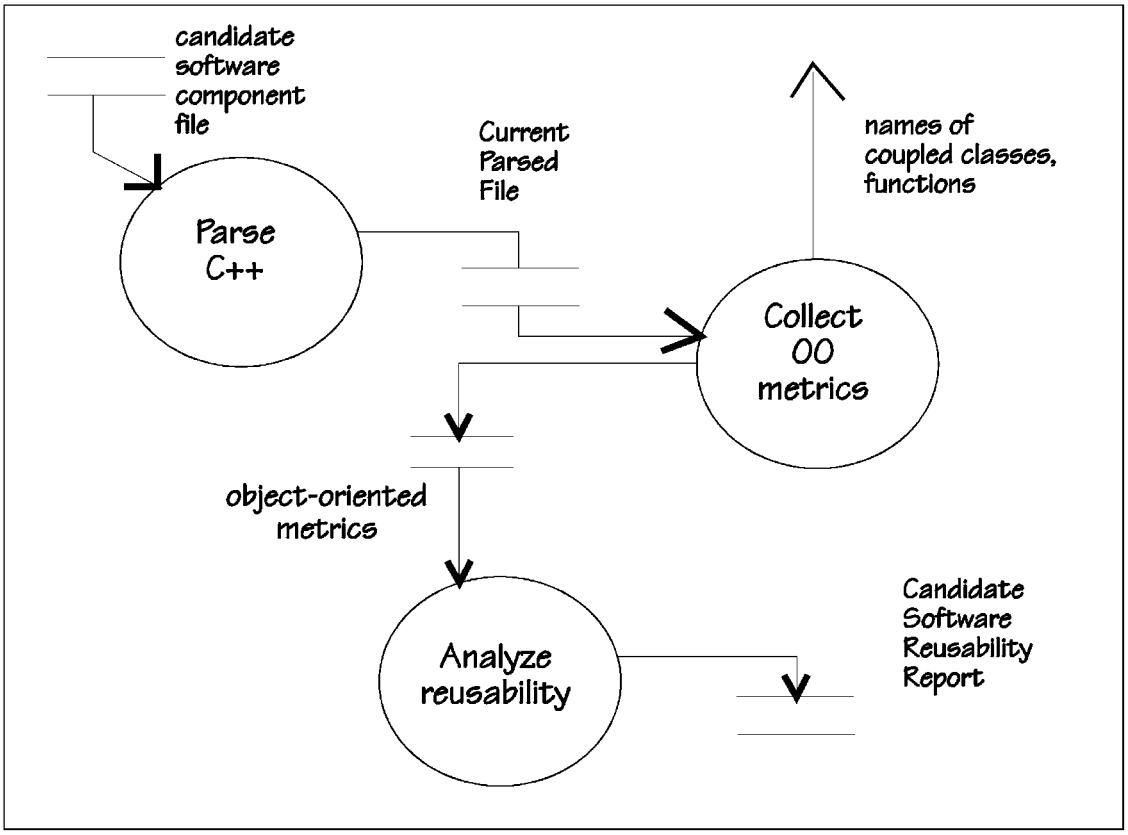


Figure 39. PATRicia System Level 2 Data Flow Diagram -- the Metrics Analyzer Tool

6.2.1.2 Collect OO Metrics

The Collect OO Metrics transform has as input the current parsed file, which is an abstract syntax tree produced by the Brown University C++ Parser. The Collect OO metrics transform analyzes the abstract syntax tree information, and uses this information to calculate several object-oriented metrics. The metrics calculated are listed in Figures 22 and 23 in Chapter V. A few metrics are calculated here by use of the PCMETRIC™ tool. These metrics are Number of Children, average number of commented methods, and average number of comment lines per method. Additionally, McCabe's complexity values for member functions that are provided by the PCMETRIC™ tool are used in the calculation of the WMC metric, sum of the static complexities of all local methods. All other metrics are calculated using information from the abstract syntax

tree. Outputs from the Collect OO Metrics transform are the various object-oriented metrics, in a file, and the names of friend classes and functions, which are used separately by the **CHRiS** module.

6.2.1.3 Analyze Reusability

Input to the Analyze Reusability transform are object-oriented metrics for each class or class hierarchy. First, each metric is mapped to a value between 0 and 1.00, using an empirical threshold-based function (thresholds used are from Lorenz and Kidd [78]). Then, for each class, the quality factors shown in Figure 22 of Chapter V are calculated as a weighted sum of the mapped metric values. Output from the Analyze Reusability transform is a candidate software reusability report.

6.2.2. The Brown University C++ Parser

The Brown University C++ parser provides an abstract syntax tree for a C++ file. It is not a compiler in that it does not produce object software. An example of the output of the Brown University C++ parser is shown in Figure 40. This output represents part of the parse of a function. The Brown University C++ parser produced a parse of the C++ software of the function GetY--the parse is represented as keywords in ASCII. For example, the FctDecl keyword signals that the following is the beginning of the function declaration. The ReturnStmt keyword signifies that a return statement in C++ has been found. Indentation within the output file corresponds to hierarchy levels. The hexadecimal numbers shown at the left hand side of the file are unique identifiers assigned to each node in the parse tree. Much of the challenge of the development of the **Metrics Analyzer** tool was to handle parsing of the abstract syntax tree file for all possible cases.

The Brown University C++ parser handles most C++ constructs; however, there are several it will not handle. It has difficulty compiling some XWindows files, and some common GNU

```
   150a00          Object_Function GetY  ### 9 Line 39 of file "li.C"
*  14e700            Scope 5         ### 8 Line 27 of file "li.C"
*  150c00            Type_MemberFunction         ### 4 Line 36 of file "li.C"
*  b7940             ProtectSpec public      ### 0
*  b7640             LinkageSpec C++       ### 0
*  b7b00             StorageSpec gfunction### 0
   13e680            FctDecl         ### 5 Line 39 of file "li.C"
*  150a00              Object_Function GetY        ### 9 Line 39 of file "li.C"
   13e800             CtorInitList   ### 0 Line 39 of file "li.C"
   13e6c0             FctBody        ### 1 Line 39 of file "li.C"
   13e780               StmtList     ### 1 Line 39 of file "li.C"
   13e740                 ReturnStmt         ### 1 Line 39 of file "li.C"
   151c80                  BuiltinCast         ### 2 Line 39 of file "li.C"
*  b9a00                    Type_Primitive int        ### 0
   13c980                   SimpleName Y  ### 2 Line 39 of file "li.C"
*  14e200                     Object_Variable Y     ### 6 Line 29 of file "li.C"
*  150500                     Type_Address          ### 1 Line 31 of file "li.C"
   150100               Scope 4        ### 2 Line 39 of file "li.C"
*  14e700                 Scope 5      ### 8 Line 27 of file "li.C"
   152e00                 Object_Variable this          ### 6
   152f80                   Scope 7      ### 2 Line 39 of file "li.C"
*  150100                     Scope 4  ### 2 Line 39 of file "li.C"
*  152e00                    Object_Variable this      ### 6
*  150f80                   Type_Qualified     ### 2 Line 31 of file "li.C"
*  b7900                    ProtectSpec none  ### 0
*  b7640                    LinkageSpec C++ ### 0
*  b7b80                    StorageSpec argument      ### 0
*  b75c0                    EmptyExpr### 0
*  152f80               Scope 7        ### 2 Line 39 of file "li.C"
   14d180              ExprList        ### 1
*  b75c0                EmptyExpr    ### 0
*  b7f80              KeyDeclSpec inline      ### 0
*  b7f00              KeyDeclSpec not_virtual         ### 0
```

Figure 40. Abstract Syntax Tree -- Output of the Brown University C++ Parser

library files (the version of the C++ parser used in the **Metrics Analyzer** will not handle the GNU include file iostream.h without changes). It will not handle DOS-style code constructs such as "far" pointers. Due to problems in getting some to compile with the Brown University C++ parser, the **Metrics Analyzer** tool was modified to operate on partial parses of source code files. As long as the actual code of interest is parsable, the **Metrics Analyzer** will work.

6.2.3. Description of Operation

When the understood candidate component is presented to the Collect OO metrics function, it is first parsed by the Brown University C++ parser. This produces an abstract syntax tree from which the Collect OO metrics function is able to gather structural information such as calls from one member function to member functions of other classes or to global functions, calls from one member function to attributes of other classes or to global variables, number of times a class is used as an abstract data type, number of member functions per class, number of attributes per class, etc. In fact, many of these quantities derived from the abstract syntax tree correspond fairly directly to low-level object-oriented metrics. Others can be used to calculate various object-oriented metrics fairly easily. After the Collect OO metrics function has collected and stored the structural information derived from the abstract syntax tree, then it calculates several object-oriented metrics using that data. Then the Collect OO metrics function reads the output of the PCMETRIC™ tool to collect various other metrics. The primary metrics that are collected from the PCMETRIC™ tool are the class hierarchy chart, the McCabe's cyclomatic complexity number for member functions, and counts of lines of code and comments for member functions.

The PCMETRIC™ tool in its current form does not produce the desired output without user intervention. This is due to the fact that most of the **PATR**icia system runs under Linux, whereas the PCMETRIC™ tool is a DOS utility. However, the PCMETRIC™ tool is capable of running in the Linux DOS emulator. The PCMETRIC™ tool can be eventually replaced with separate software, which will result in the complete automation of the **Metrics Analyzer.**

```
REUSE-IN-THE-CLASS METRICS

Metrics for class TAppWindow, with node number 976d00
 MODULARITY:
  COUPLING:
   MPC is 26
   AVC is 18
   DAC_coupling is 17
   num_friend_classes is 0
   num_friend_functions is 0
  COHESION:
   LCOM1 is 247
   LCOM1_without_constructor_destructor is 247
   LCOM2 is 247
   LCOM2_without_constructor_destructor is 247
   LCOM3 is 21
   LCOM3_without_constructor_destructor is 21
   LCOM4 is 21
   LCOM4_without_constructor_destructor is 21
 INTERFACE:
   num_public_functions is 23
 DOCUMENTATION:
   avg_num_of_comment_lines_per_method is       0.83
   percentage_of_commented_methods is      47.83
   num_comments_in_class_definition is 13
 SIMPLICITY:
  SIZE:
   num_methods is 23
   num_attributes is 9
   avg_method_size_in_LOC is      19.22
   avg_method_size_in_executable_semicolons is       9.57
  COMPLEXITY OF METHODS:
   WMC is 44
```

Figure 41. Object-Oriented Metrics Report -- the Metrics Analyzer Tool

```
REUSE-IN-THE-ORIGINAL-SYSTEM METRICS

Metrics for class TAppWindow, with node number 976d00

  ABSTRACT DATA TYPE USAGE:
    num_uses_as_abstract_data_type is 4
  INHERITANCE USAGE:
    DIT is 1
    NOC is 0
  CALLS USAGE:
    number of calls to member functions in this class is 8
```

Figure 42. Reuse-in-the-Original-System Object-Oriented Metrics Report

```
REUSE-IN-THE-CLASS

Reusability results for class wxbBrush with node_number 879400
 MODULARITY     is  0.93
    COHESION     is  0.85
    COUPLING     is  1.00

 INTERFACE      is  1.00

 DOCUMENTATION  is  0.00

SIMPLICITY     is  1.00
    SIZE         is  0.99
    COMPLEXITY   is  1.00


REUSABILITY       is  0.73
```

Figure 43. Reusability-in-the-Class Reusability Quality Metrics Report

```
REUSE-IN-THE-HIERARCHY

Reusability results for class wxObject with node_number 769200
 MODULARITY     is  0.99
       COHESION     is  0.99
       COUPLING     is  1.00
INTERFACE      is  1.00
DOCUMENTATION  is  0.50
SIMPLICITY     is  0.99
    SIZE         is  0.99
    COMPLEXITY   is  1.00

HIERARCHY      is  1.00

REUSABILITY       is  0.87
```

Figure 44. Reusability-in-the-Hierarchy Reusability Quality Metrics Report

After the metrics are gathered from PCMETRIC™, the Analyze reusability function uses those metrics, and the object-oriented metrics calculated by the Collect OO metrics function using structural information from the abstract syntax tree to calculate various reusability metrics. The Reusability-in-the-Class quality factors are calculated, and an overall number for Reusability-in-the-Class is calculated. The Reusability-in-the-Hierarchy quality factors are calculated, and an overall number for Reusability-in-the-Hierarchy is calculated. See Chapter V for a discussion of these reusability hierarchies, and the metrics that combine to form reusability quality factors.

6.2.4. Reports Produced by the Metrics Analyzer Tool

The **Metrics Analyzer** produces five reusability reports: a list of all the object-oriented metrics in each class, a Reusability-in-the-Class report, a Reusability-in-the-Original-System report, a Reusability-in-the-Hierarchy report, and a report that contains all the structural information that was derived by parsing the abstract syntax tree file.

Figure 41 contains an example of the object-oriented metrics report. In this report, MPC is Message Passing Coupling, AVC is Attribute/Variable Coupling, and DAC_coupling is Data Abstraction Coupling. The various LCOM values relate to the discussion on the multiple definitions of LCOM in the literature (see Chapter V). LCOM1 is the revised Chidamber and Kemerer LCOM definition [26], with inheritance. LCOM2 is the revised Chidamber and Kemerer LCOM definition, without inheritance. LCOM3 is the Li and Henry LCOM definition [75], with inheritance. LCOM4 is the Li and Henry LCOM definition, without inheritance. WMC stands for weighted methods per class, and is the sum of the McCabe's cyclomatic complexity numbers for each of the member functions of the class. The metrics in this report are grouped according to the quality factor they are said to affect. However, the resulting values of each quality factor are not shown in this report.

An example of a Reusability-in-the-Original-System object-oriented metrics report is shown in Figure 42. An example of a Reusability-in-the-Class reusability quality metrics report is shown in Figure 43. In this report, separate values for reusability quality metrics are provided for quality factors and for quality sub-factors such as cohesion and coupling. Figure 44 is an example of a Reusability-in-the-Hierarchy report. The report is similar to that of the Reusability-in-the-Class report, with the exception of the Hierarchy quality factor.

Chapter VII

RESULTS

7.1. Description of Domain

The initial test domain for the **PATR**icia system was the domain of graphical user interfaces (GUIs). This was chosen for multiple reasons. First, the domain is large enough to require a fairly complex knowledge-base. Second, several C++ GUI packages were easily available. Third, the area of graphical user interfaces, and the C++ software used to implement a graphical user interface is a real world domain, with real software. Fourth, there are plans to upgrade the **PATR**icia system with a graphical user interface of its own and this allowed an evaluation of potential graphical user interface packages for that purpose.

Since the **PATR**icia system is designed to measure the concepts and functions which are implemented in object-oriented software, it is critical to know how accurate and effective that assessment is. The most effective way to carry out that evaluation is through the use of knowledgeable software developers who are experienced in the domain of interest. Using their insight and experience to identify concepts and comparing those concepts with the **PATR**icia system's evaluation will provide an objective view of the tool's capabilities. These experiments concentrated upon two critical aspects of the tool's performance: the ability to identify concepts and the ability to predict reusability.

7.2. Program Understanding Experiment

7.2.1. Description of Experiment

The purpose of the program understanding experiment was to compare the concepts identified by the members of the evaluation team, each of whom were knowledgeable in the area of C++ and graphical user interfaces, to those identified by the **CHR**i**S** tool within the **PATR**icia system. In this experiment, classes taken from three independent C++ graphical user interfaces were examined both by knowledgeable software developers and the **PATR**icia system. The experiment was conducted in two phases. The first phase was to identify an agreed-upon list of concepts that occurred in the selected software classes. The second phase was to compare the tool's output with those concepts. A list of the domain experts who contributed to the study and their qualifications is provided in Appendix D.

The C++ graphical user interface packages used in the experiments were the GINA package [8], the wxWindows package [102], and the Watson XWindows package [111]. These were chosen because they were widely used packages that are representative of the software that could normally be encountered in the GUI area.

7.2.2. Phase I--Determination of Concepts Handled by a Class

The objective of Phase I of the program understanding experiment was to identify the concepts (primarily functionality) handled by each of a set of classes chosen from three independent C++ graphical user interface packages. The five highly experienced C++ and graphical user interface experts were given a list of standard concepts, with concept definitions, that represented the list of concepts available in the **CHR**i**S** knowledge-base at that time. They were given the opportunity to add concepts or to redefine concepts if necessary. Each reviewer was also given a notebook containing C++ source code for classes chosen from the three C++

GUI packages. The reviewers then examined the source code, and listed for each class the concepts that were covered by that class.

It was estimated that this procedure would require five to six hours of work from each reviewer; however, the reviewers reported that it actually required from twelve to fifteen hours of work. One reviewer had personal problems and was unable to complete his task. The directions given to the reviewers, and questionnaires used by the reviewers are provided in Appendix E. At the end of Phase I, a merged list of the concepts for each class identified by the reviewers was prepared. This merged list was used in the formation of questionnaires in Phase II of the experiment. An example questionnaire containing concepts for a class hierarchy from the GINA system is shown in Appendix E.

7.2.3. Phase II--Mapping of Concepts Identified by the PATRicia System

The purpose of Phase II was to compare the output of the **CHRiS** tool of the **PATR**icia system to the merged concept list identified by the reviewers for each class. An additional person was chosen to replace the reviewer who had to leave the experiment before the end of Phase I. Thus, five domain experts were given the merged list of concepts for each class that was identified by the reviewers in Phase I. They were also given a notebook containing the reports generated by **CHRiS** for each class selected from three independent C++ and GUI packages.

The reviewers listed concepts as being matched exactly or partially by the concepts in the **CHRiS** report. They listed any concept generated by **CHRiS** that did not match any concepts in the merged list of concepts separately as this was an indication of overgeneration, and these are spurious concepts.

It was estimated that this procedure would take the reviewers each from two to three hours of work, but it required closer to five hours each. The directions given to the reviewers, and questionnaires used by the reviewers are provided in Appendix F.

7.2.4. Metrics for Validation

The results of the program understanding experiment were evaluated for recall, precision, and overgeneration. Recall is a measure of completeness of concept generation. Precision is a measure of the accuracy of concept generation. Overgeneration is a measure of spurious generation of concepts.

Metrics for recall, precision, and overgeneration were adapted from metrics that are used in the area of Information Extraction [34][35].

The metrics that were employed in this experiment are defined as follows:

POS (possible)--the total number of concepts identified by the reviewers for a class

ACT (actual)--the number of concepts either matched or partially matched plus the number of spurious concepts

correct--the number of concepts identified by the reviewers that were matched exactly by the output of the **CHRiS** tool

partial--the number of concepts identified by the reviewers that were partially matched (matched by similar concepts) by the output of the **CHRiS** tool

The metrics for recall were:

recall = (correct + partial X 0.5) / POS

matched recall = correct/POS

The metrics for precision were:

precision = (correct + partial X 0.5) / ACT

matched precision = correct/ACT

The metrics for overgeneration were:

overgeneration #1 = spurious/POS

overgeneration #2 = spurious/ACT

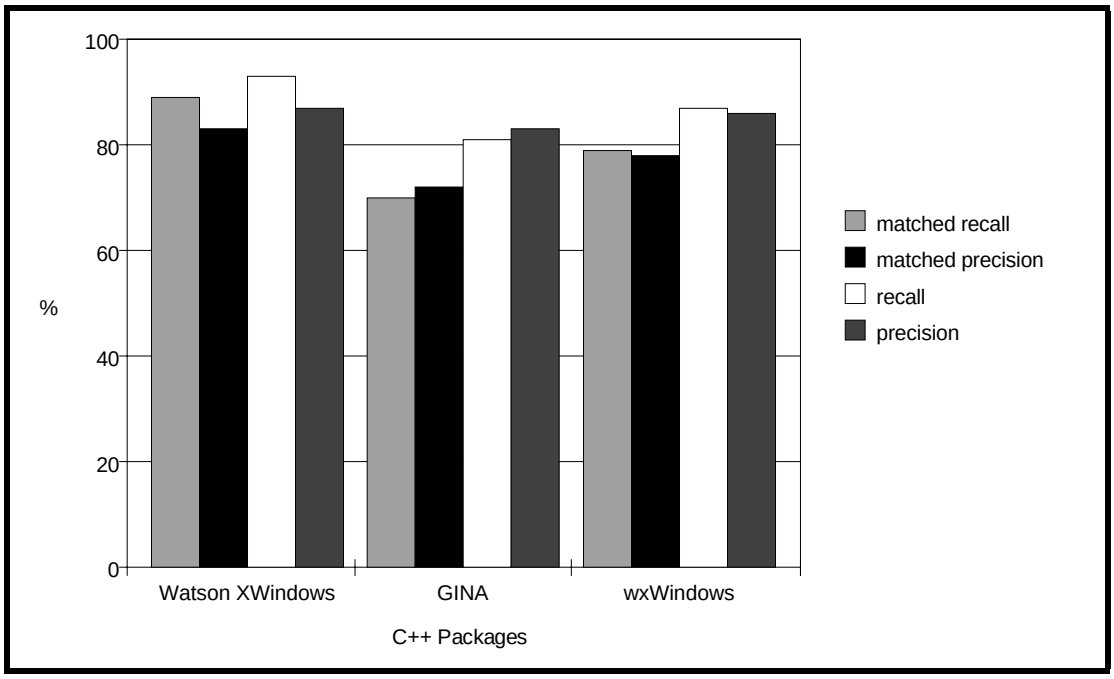


Figure 45. Recall and Precision Overall

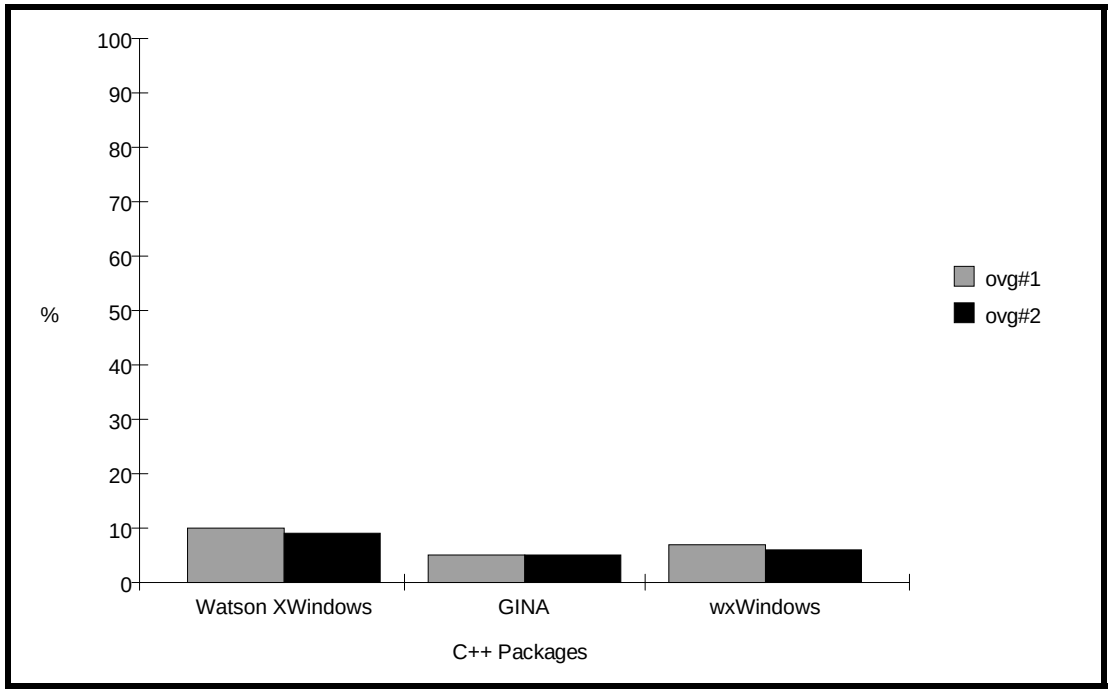


Figure 46. Overgeneration Overall

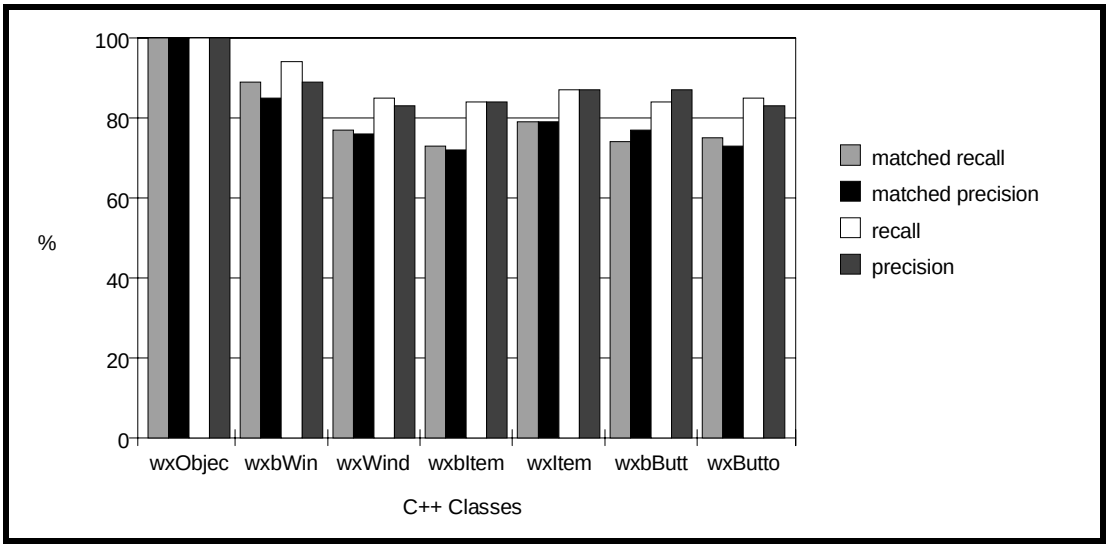


Figure 47. Recall and Precision for a wxWindows Hierarchy

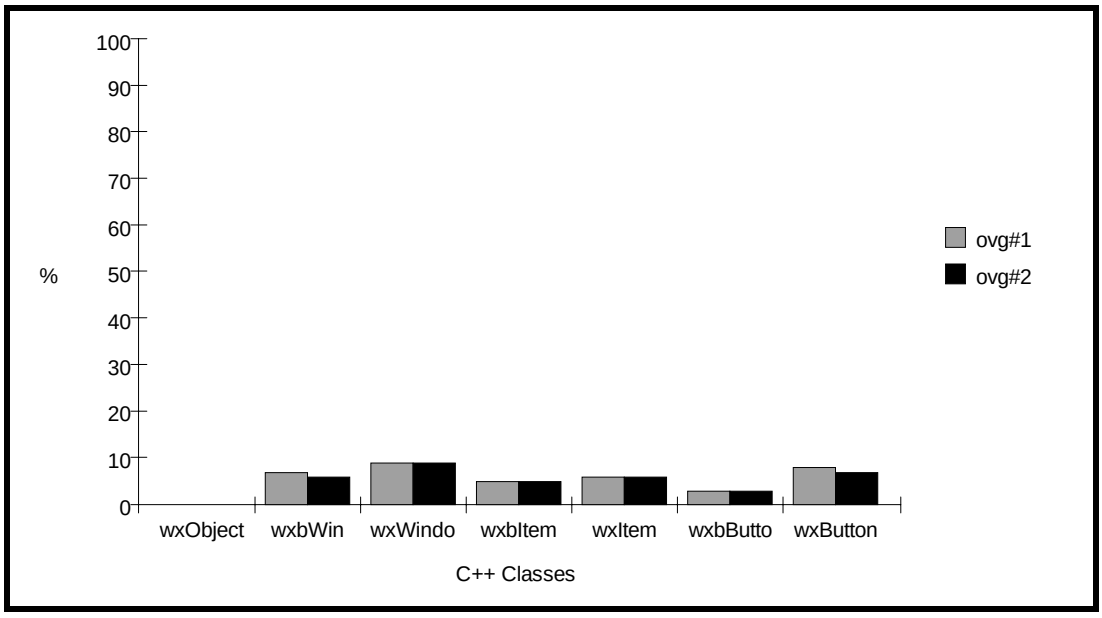


Figure 48. Overgeneration for a wxWindows Hierarchy

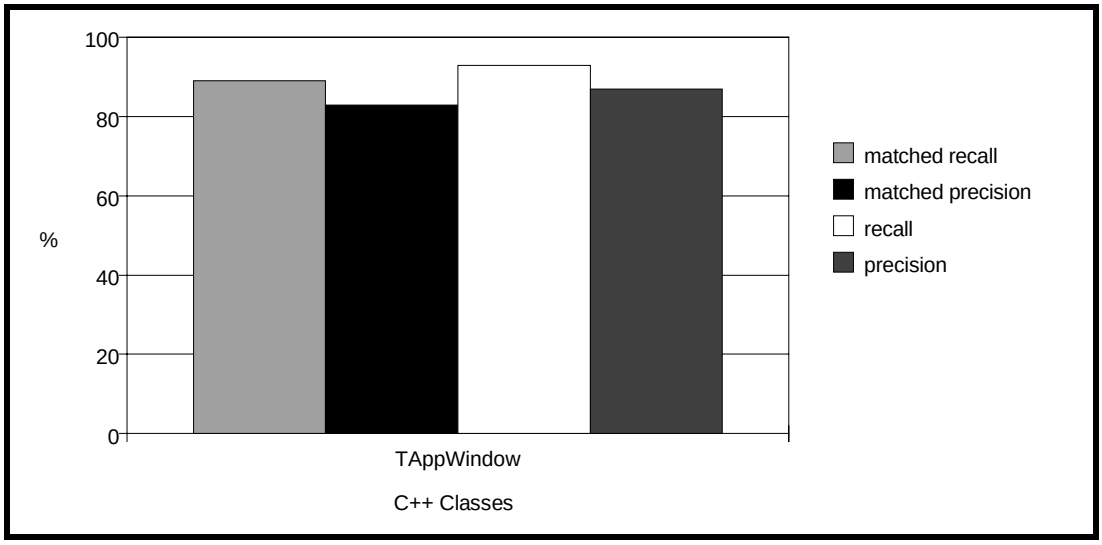


Figure 49. Recall and Precision for Watson XWindows

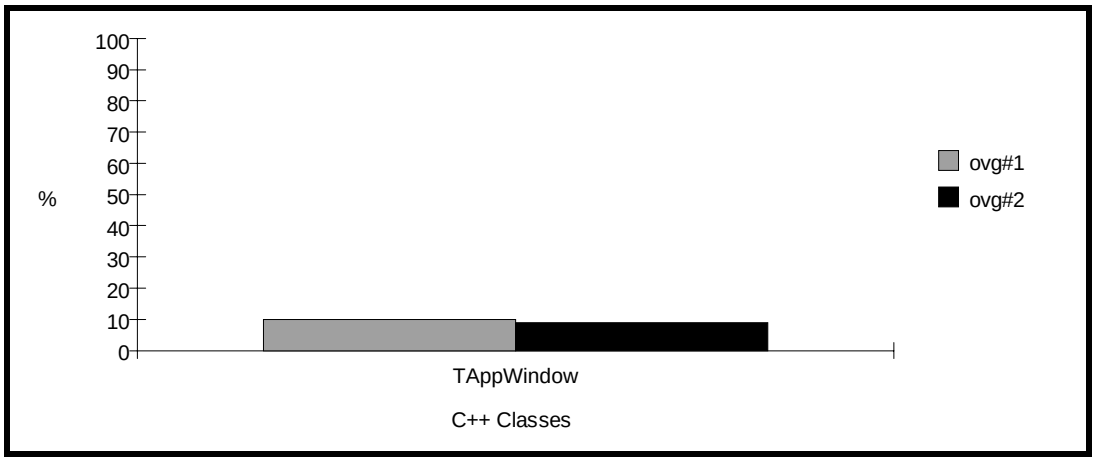


Figure 50. Overgeneration for Watson XWindows

7.2.5. Analysis of Results

The overall results for recall, precision, and overgeneration for the three different GUI packages are shown in Figures 45 and 46. An early indication can be obtained by noting that

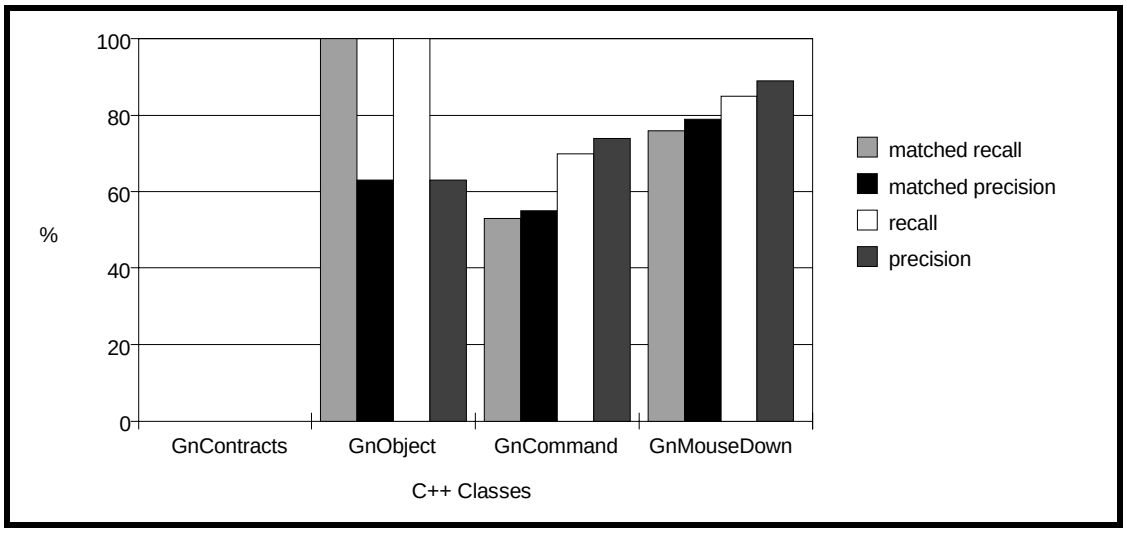


Figure 51. Recall and Precision for a GINA Hierarchy

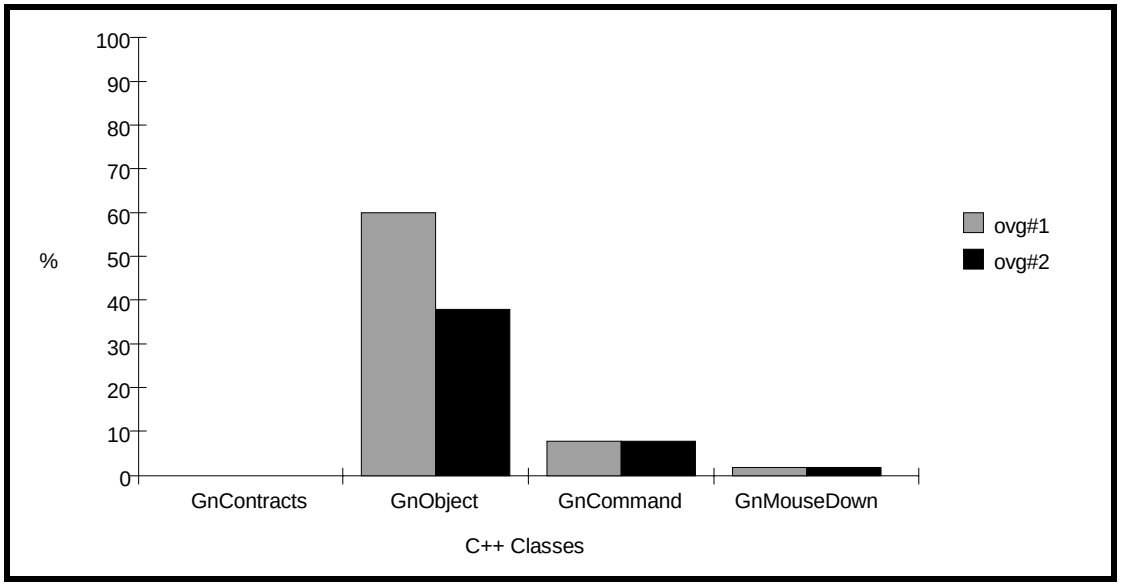


Figure 52. Overgeneration for a GINA Hierarchy

matched recall and matched precision, which are the more strict versions of the recall and precision measurements, are always 70% or above, and that overgeneration (of both categories) is under 10%.

A look at the individual software packages is shown in Figures 47 through 52. Within the

wxWindows package, matched recall or matched precision was greater than 72%, and overgeneration was under 10%.

In the Watson XWindows package, all processing is performed within a single class, and the class TAppWindow represents the entire package. Recall and precision for TAppWindow are above 80%, while overgeneration is under 10%.

In the GINA package, GnContracts had no graphical user interface concepts, so recall, precision, and overgeneration are arbitrarily shown as zero. The very high numbers for recall and for overgeneration in the class GnObject are artifacts of the fact that the class had only a single graphical user interface concept**. CHRiS** generated another concept that some reviewers counted as a spurious concept, and others did not. The classes GnCommand and GnMouseDownCommand had a larger number of graphical user interface concepts, and so the metrics for those classes approach those seen elsewhere.

It should be noted that the GINA package, which was well-designed from a code standpoint, had comments and identifiers with unusual wording which were somewhat hard to understand. The GINA package was designed and implemented at the German National Research Institute for Computer Science, and the people who implemented it were thus probably not native English speakers. (There was also a very small number of German comments in the code). Even with this potentially nonstandard wording of English comments, the operation of **CHRiS** still proved satisfactory.

7.3. Reusability Analysis Experiment

7.3.1. Description of Experiment

The purpose of the reusability analysis experiment was to determine how closely the **Metrics Analyzer** tool's predictions of Reusability and the reusability quality factors such as Modularity (cohesion and coupling), Interface, Documentation, and Simplicity (size and method complexity) matched reality.

In this experiment, seven knowledgeable C++ software developers were given a set of questions to answer and criteria to evaluate when rating classes and hierarchies for reusability. The directions and questionnaires that were given to the reviewers are provided in Appendix G.

The members of the evaluation team were also given source code from three independent C++ graphical user interface packages. They answered questions in the questionnaires about each class, and each class hierarchy. Based on their own 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 = 75%

Fair = 50%

Poor = 25%

It was estimated that this procedure should require approximately five hours of work. However, only one reviewer was able to complete the procedure in approximately that time, while the other reviewers reported that the procedure required well over twenty hours of work. According to most reviewers, the primary reason for the difficulty was the analysis of coupling. This required a strict code inspection, that was made harder by the proliferation of macros in certain packages.

7.3.2. Analysis of Results

The results which compare an averaged combination of the reviewers' reusability quality factors ratings to those of the **Metrics Analyzer** over 18 classes are shown in Figures 53-76. Each of these figures represents a comparison of the Reusability-in-the-Class quality factors hierarchy.

As can be seen by the figures, the **Metrics Analyzer** did a very good job of predicting both individual quality factors, and overall reusability for each class. In no case did the **Metrics Analyzer** predict a class as reusable that the reviewers had rated as not reusable. The reverse was not true, and in four cases the **Metrics Analyzer** rated a class as not reusable, when the reviewers had rated it as reusable. This occurs in the predictions for the classes wxbWindow (see Figure 59), wxbItem (see Figure 61), wxMouseEvent (see Figure 70), and wxWindow (see Figure 60). This was due to differences in the **Metrics Analyzer** and the evaluations team's prediction of the Interface, Documentation, and Simplicity quality factors.

As an example of how these differences occurred, the wxbWindow class will be discussed. In the case of the complexity sub-factor of Simplicity for this class, the reviewers rated the class as not complex. The class definition was quite large, with 11 attributes and 39 member functions; however, the member functions themselves were quite simple. The reviewers rated complexity low even though the class has a large number of member functions, because the member functions themselves were not complex. However, the WMC (weighted methods per class) metric that was used to measure complexity added all the complexities of each method of a class. Thus the metric assumes that complexities are additive. The WMC metric is a standard object-oriented metric; however, based on these results, a better metric might be AMC, or Average Method Complexity.

In the case of the Interface quality factor, the **Metrics Analyzer** rated it low for that class since it had 49 public member functions. The Lorenz threshold for this metric was 20, so wxbWindow was well over twice the threshold. Apparently the reviewers thought that the Interface was clean and understandable due to good, informative member function identifier names, and rated it high.

In the case of Documentation for the class, the reviewers rated the documentation quite high, whereas the **Metrics Analyzer** rated it low. In this class, the average number of comment lines per method was 0.35, and the percentage of commented methods was 8%, which are quite low numbers. However, the number of comments in the class definition was 35, which divided by the 49 methods plus 11 attributes in the class definition resulted in about a 50% comment rate, which although it is a fairly low number was still higher in that class than in most others in the package. It is possible that, even though the class was not particularly well documented in general, it received a better Documentation score since it was better documented than surrounding classes. More than likely the good identifier names were another reason why the reviewers rated this class good for Documentation.

For the classes wxbItem and wxMouseEvent, Simplicity, Documentation, and Interface are again rated low by the **Metrics Analyzer**. The problems here seems to be identical to the problems that occurred with wxbWindow in that the reviewers did not think that having a large number of uncomplex member functions made the class more complex, and apparently the reviewers considered good variable names to be very important to Interface and Documentation. In wxWindow, Interface again was rated low by the **Metrics Analyzer**. Again, the problem seemed to be that the reviewers thought good member function names were more important than the size of the interface. An obvious direction for future research would be to try to discover a way to measure the quality of variable names.

The results comparing an averaged combination of the reviewers' reusability quality factors ratings to those of the **Metrics Analyzer** over several class hierarchies are shown in Figures 69-76. In these figures a rating for Inheritance Complexity is not shown since the reviewers did not rate that quality factor separately, although they did include depth of inheritance tree in their evaluation criteria for the class. In all cases, hierarchies that the reviewers rated as acceptable were also rated as acceptable by the **Metrics Analyzer**, with a score of 50% reusability considered the minimum for acceptability. Only one hierarchy was rated as not reusable, and both the evaluators and the **Metrics Analyzer** rated this hierarchy as unacceptable.

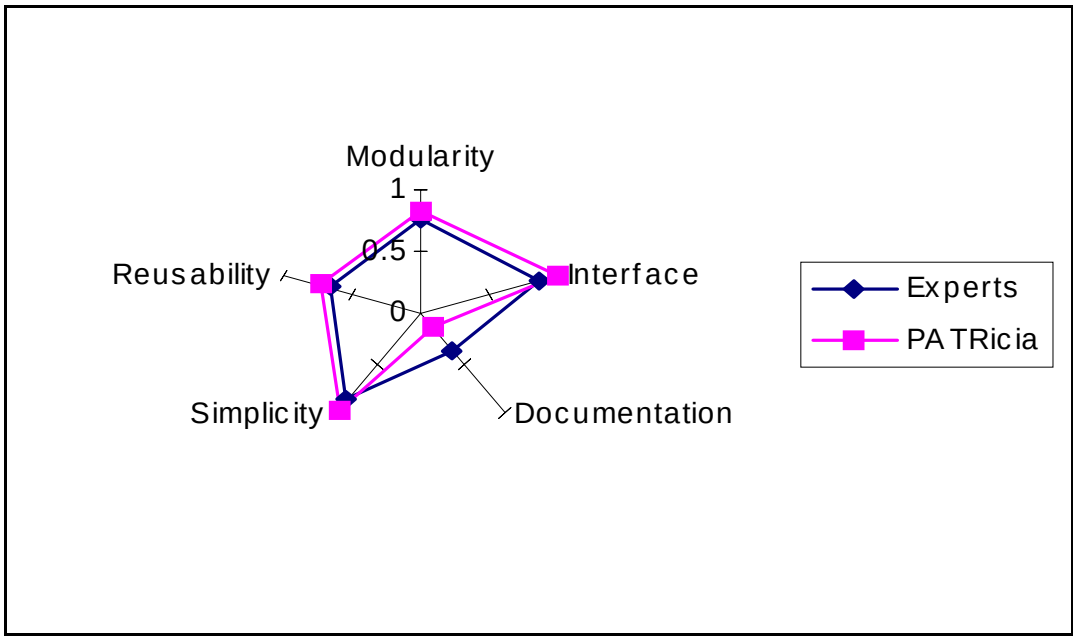


Figure 53. GnContracts

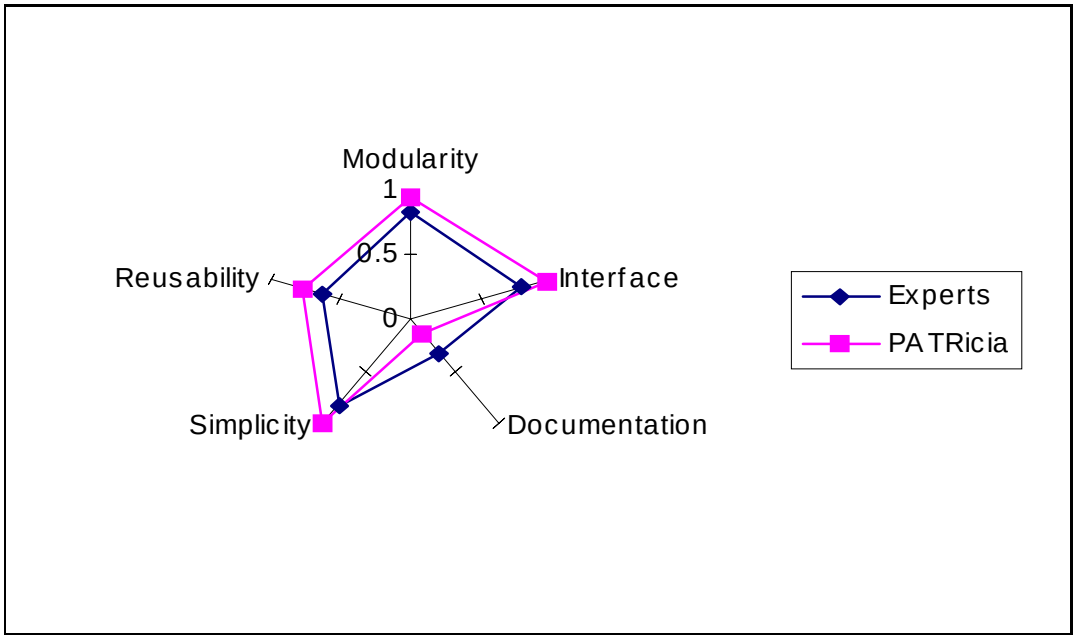


Figure 54. GnObject

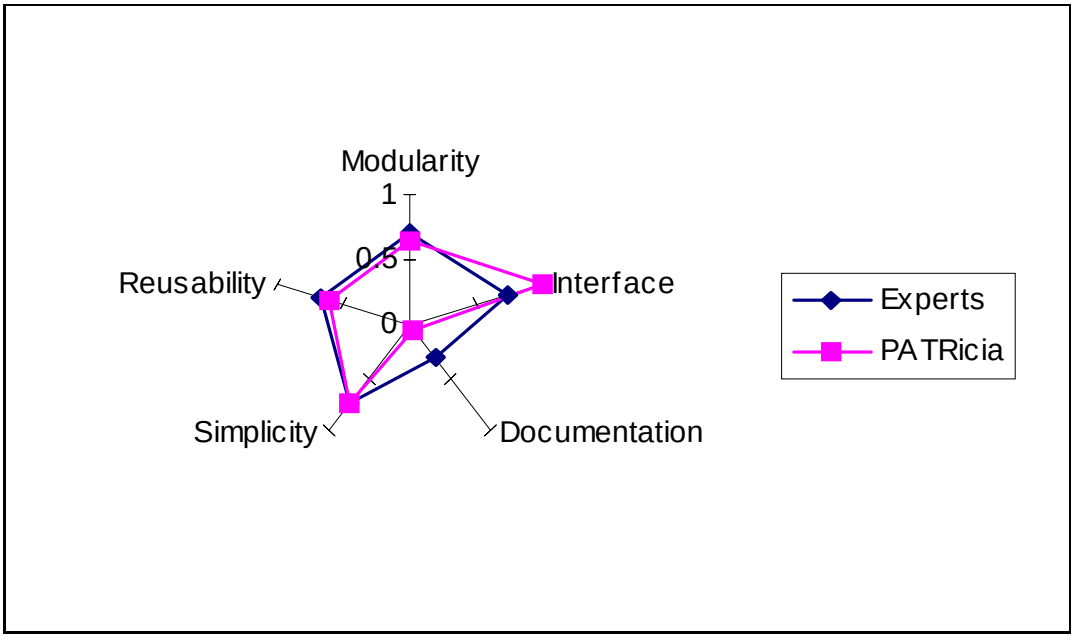


Figure 55. GnCommand

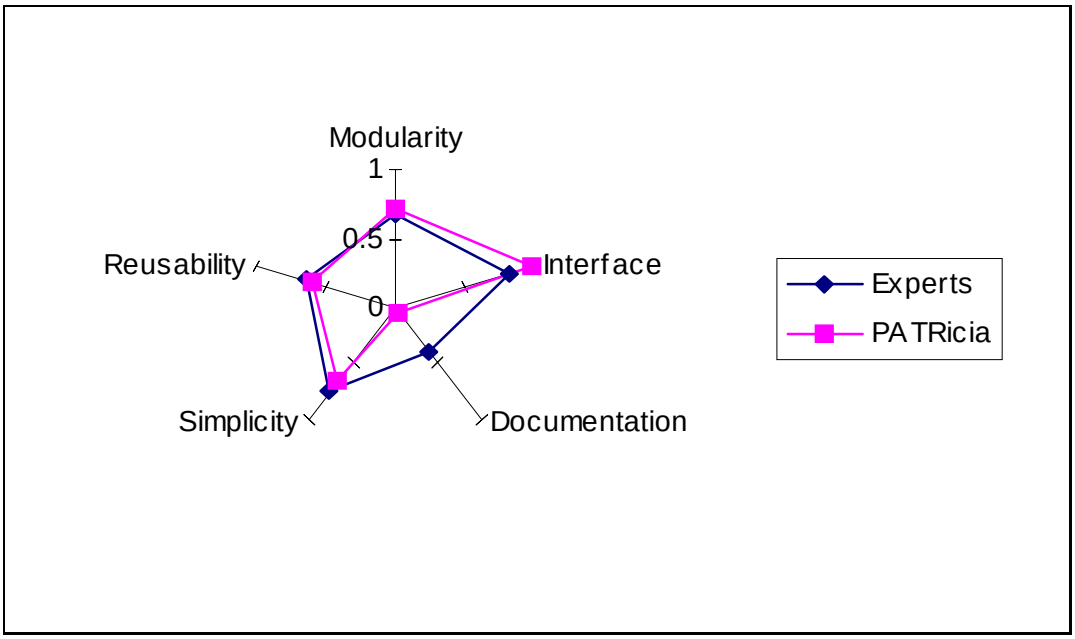


Figure 56. GnMouseDownCommand

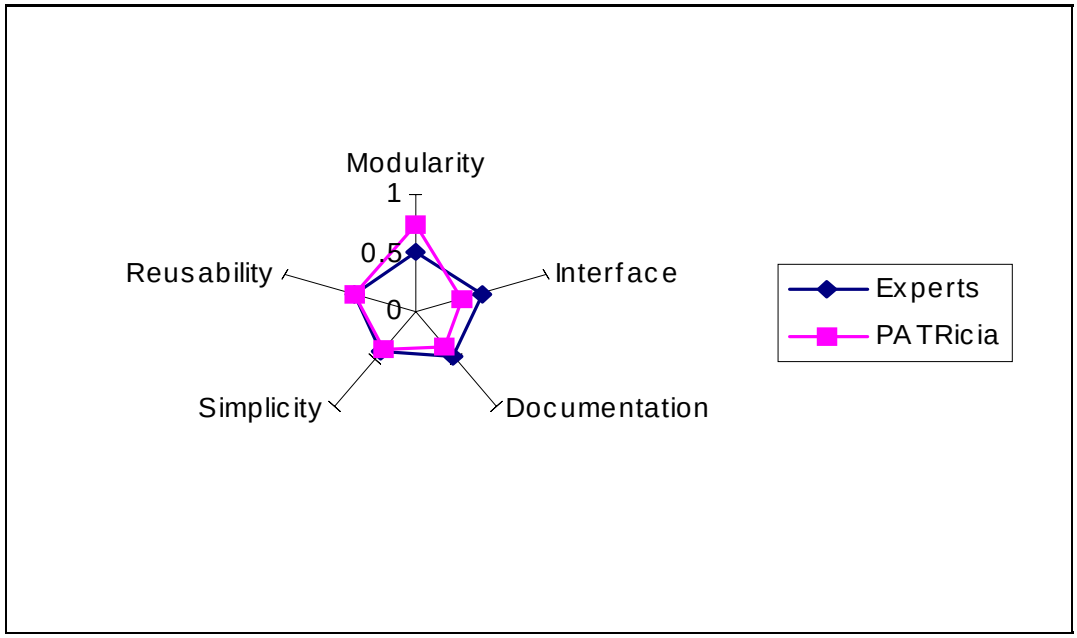


Figure 57. TAppWindow

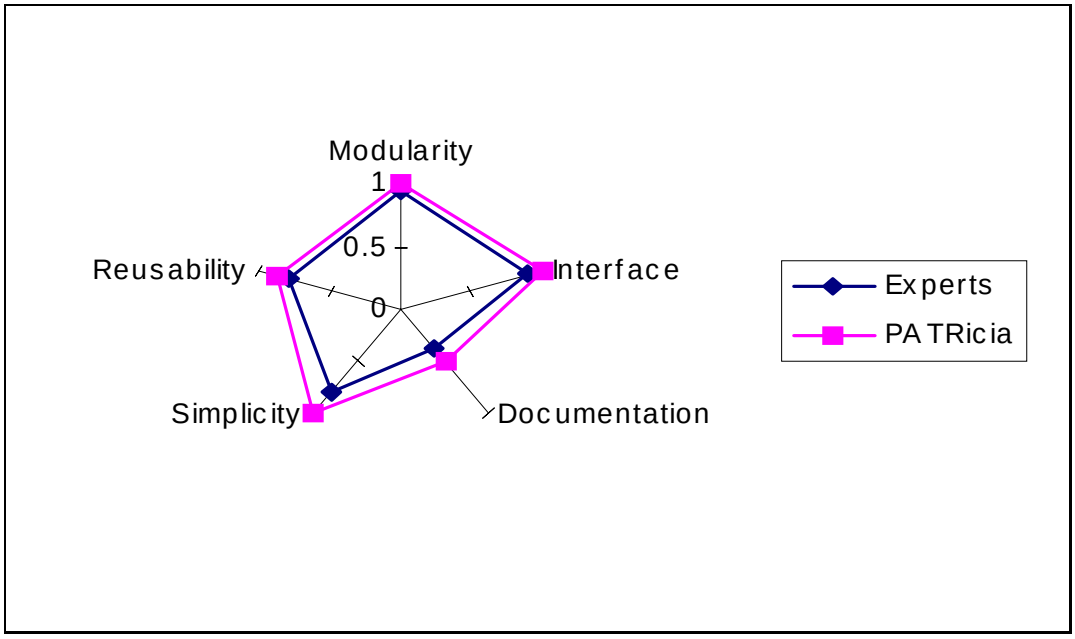


Figure 58. wxObject

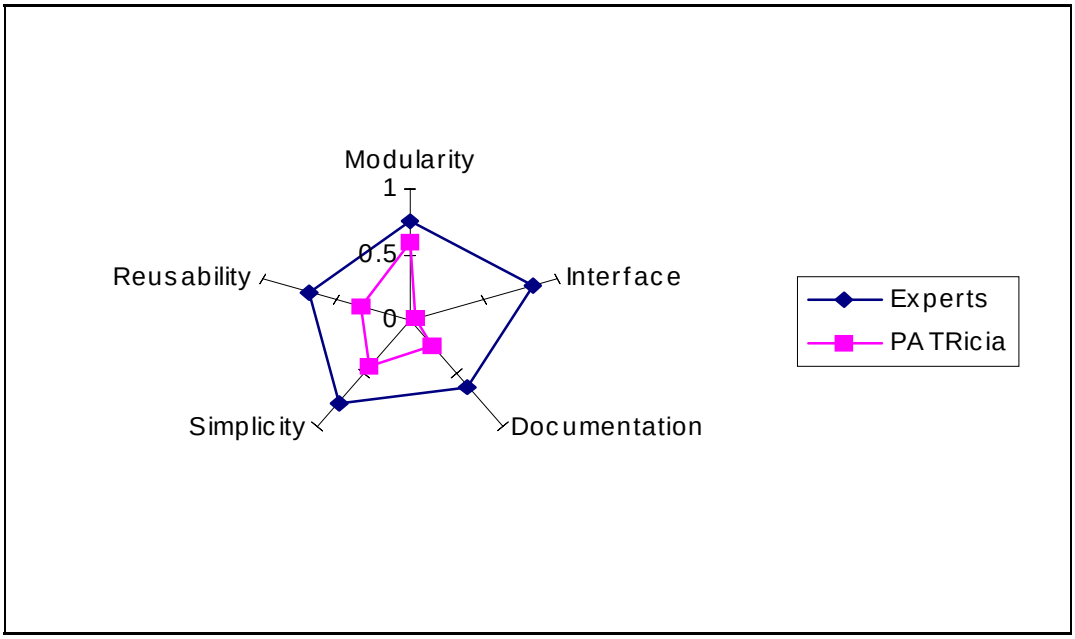


Figure 59. wxbWindow

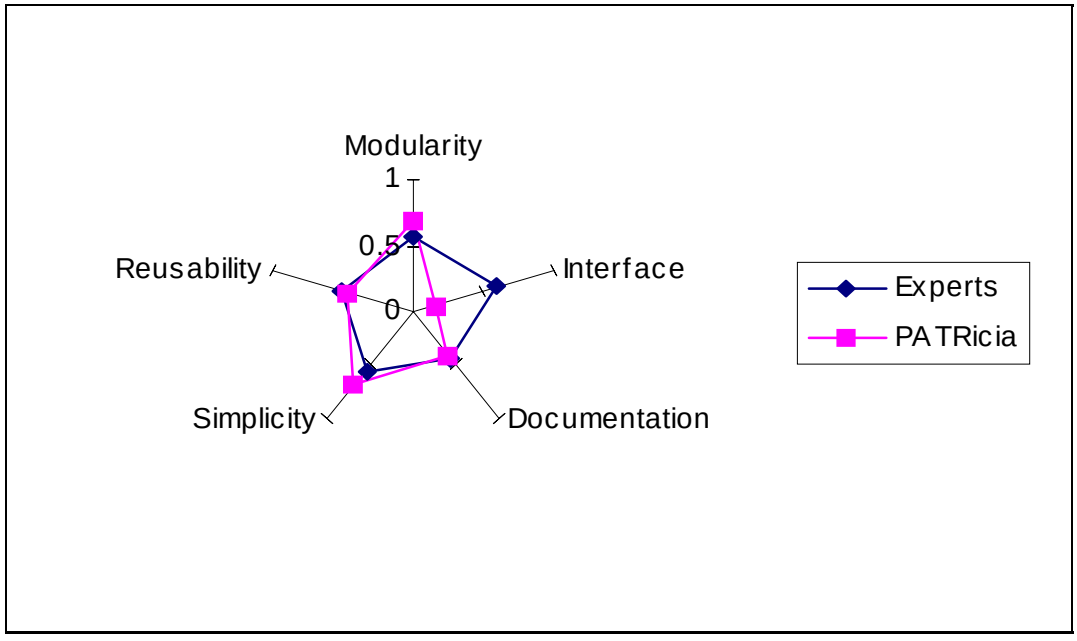


Figure 60. wxWindow

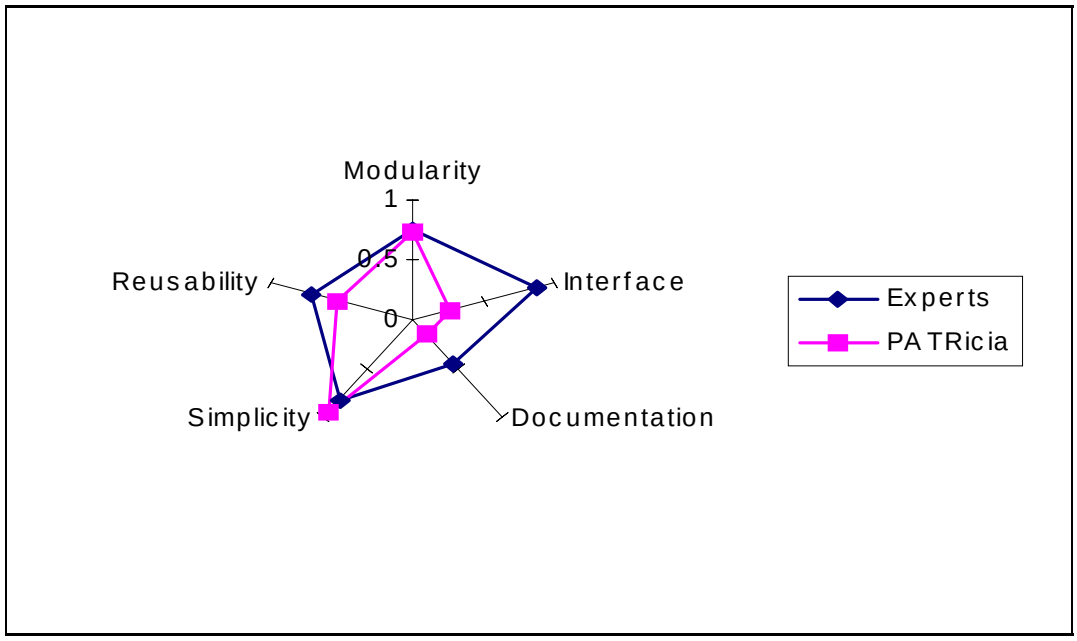


Figure 61. wxbItem

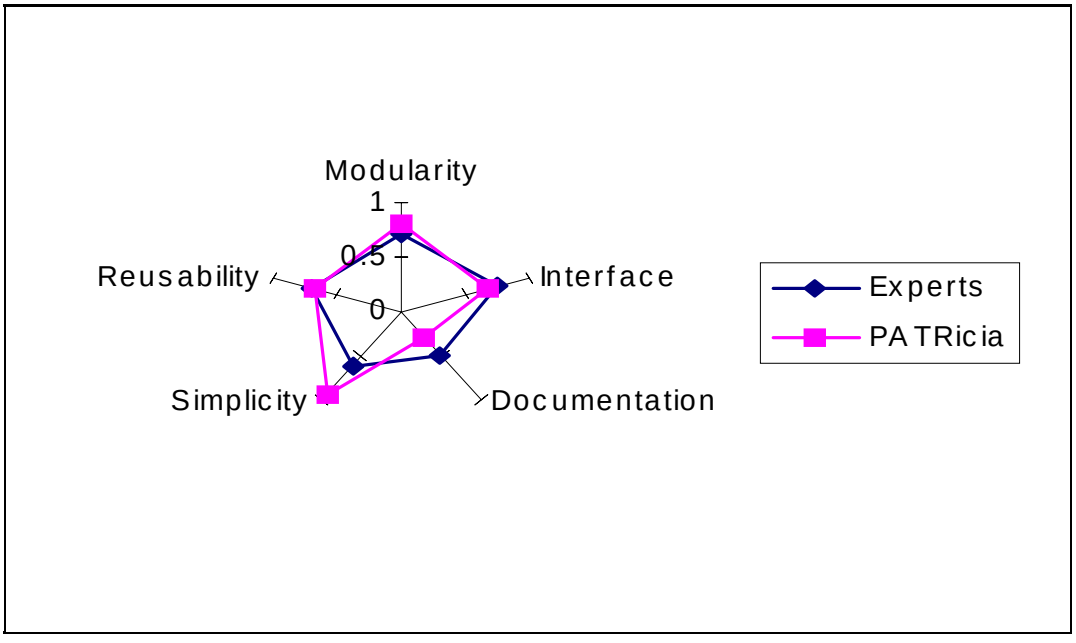


Figure 62. wxItem

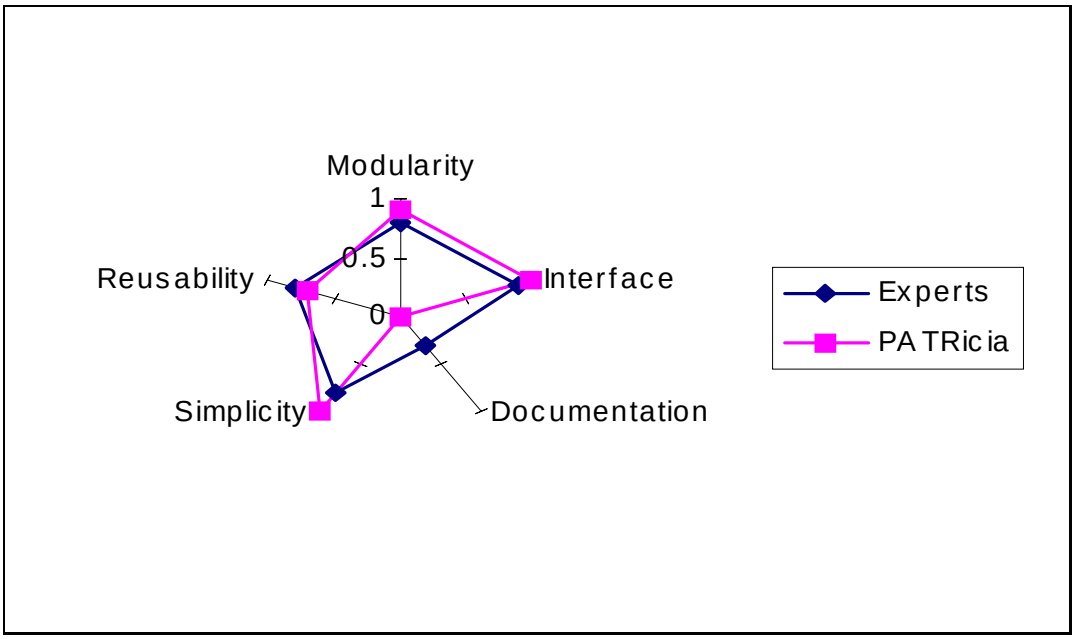


Figure 63. wxbButton

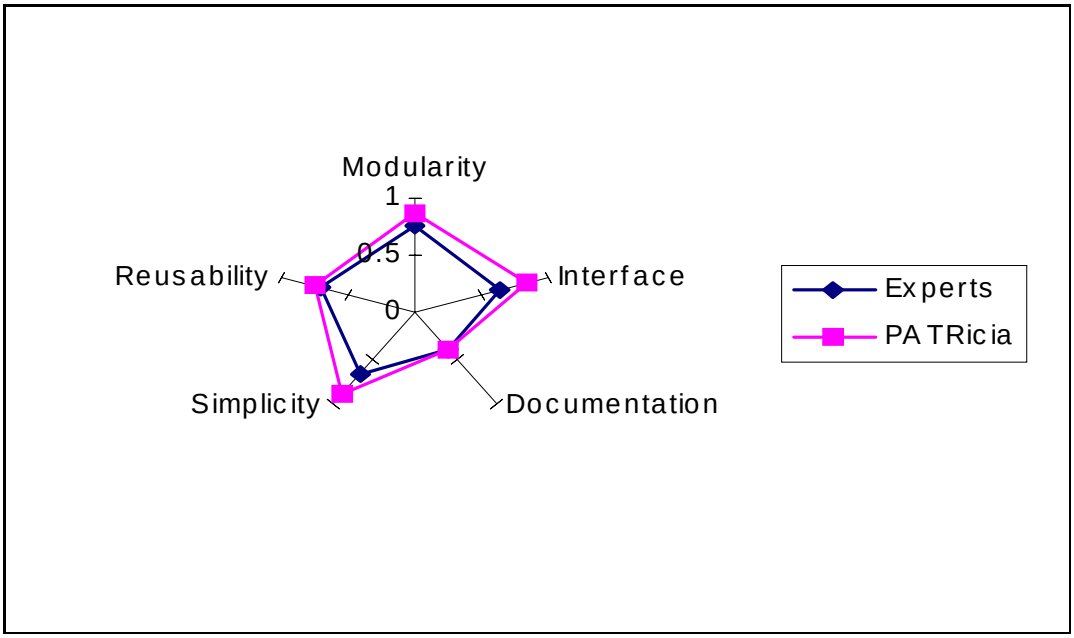


Figure 64. wxButton

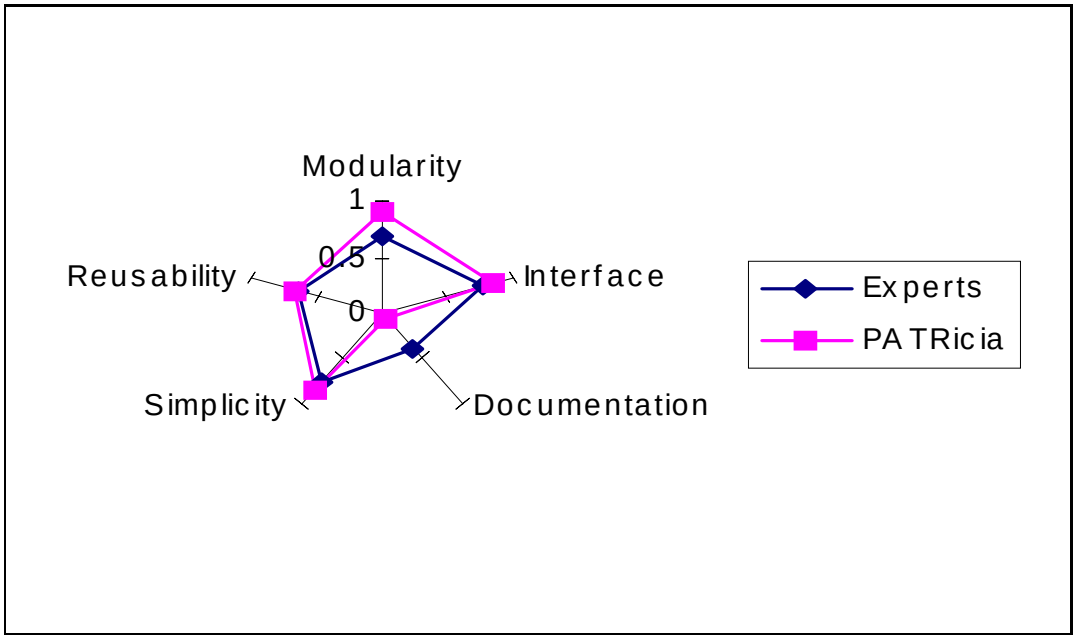


Figure 65. wxbMenu

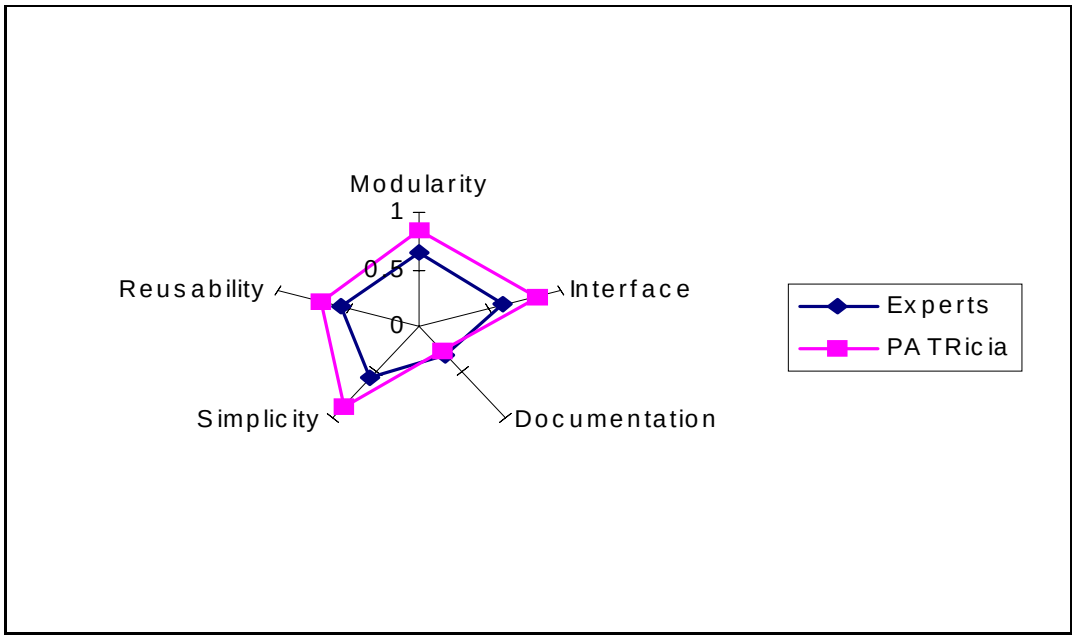


Figure 66. wxMenu

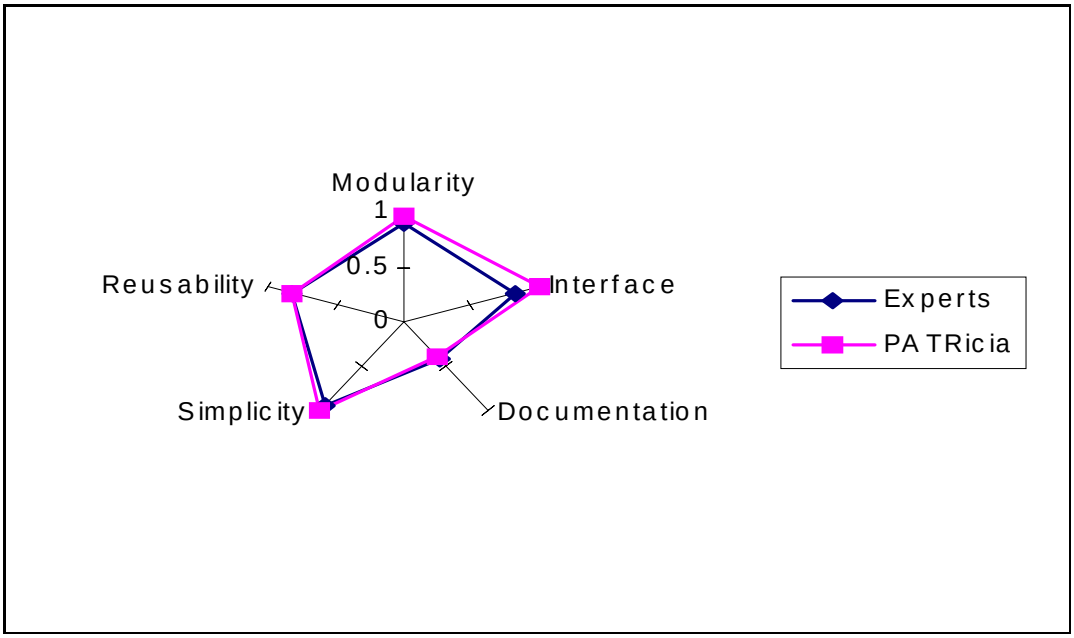


Figure 67. wxbTimer

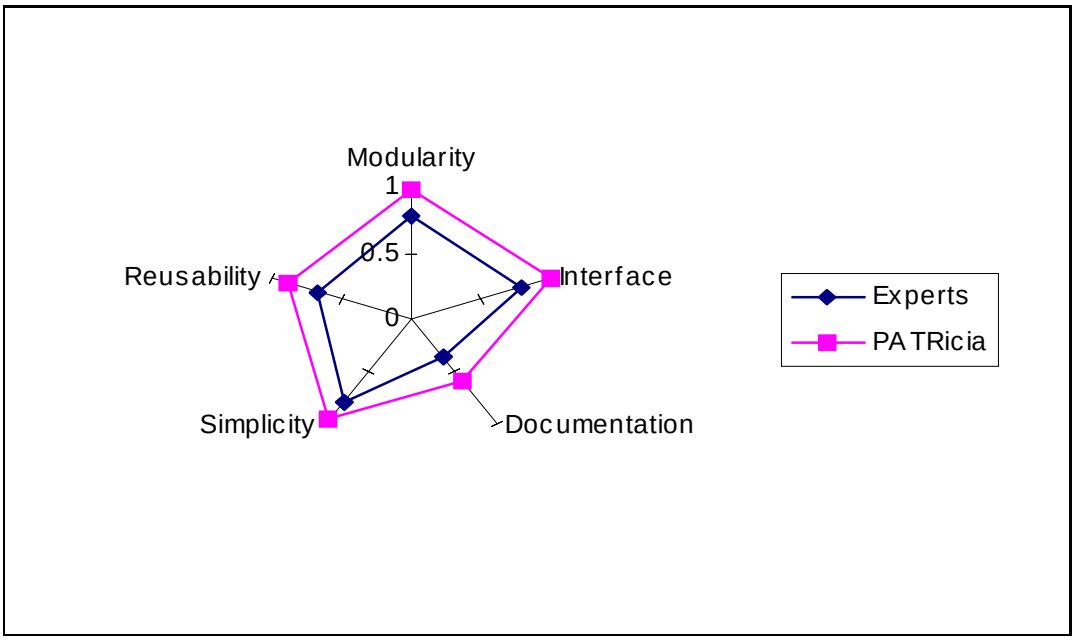


Figure 68. wxTimer

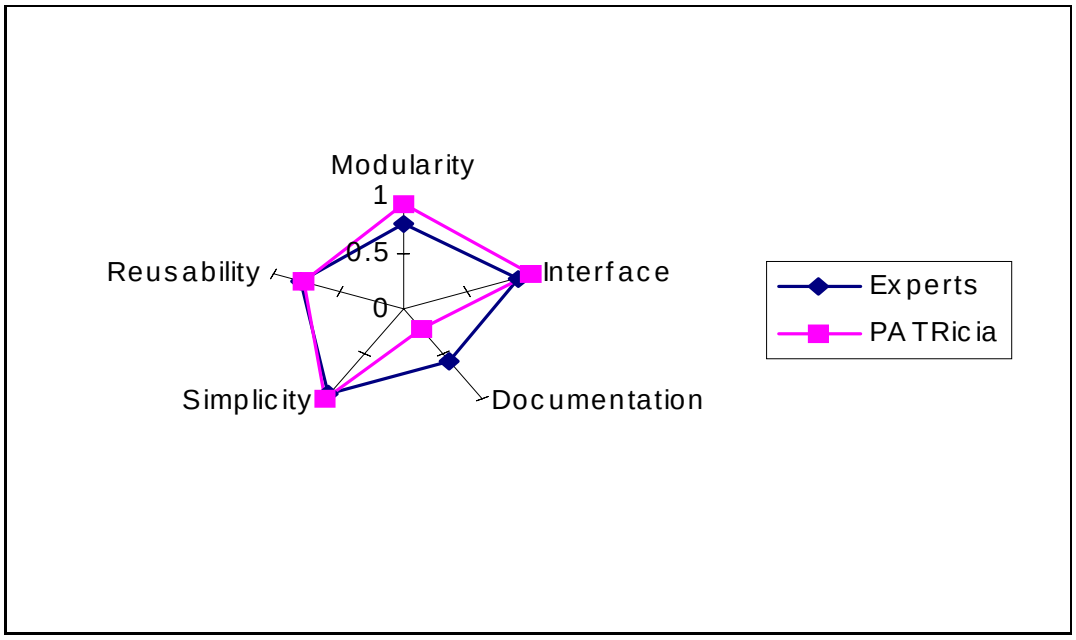


Figure 69. wxEvent

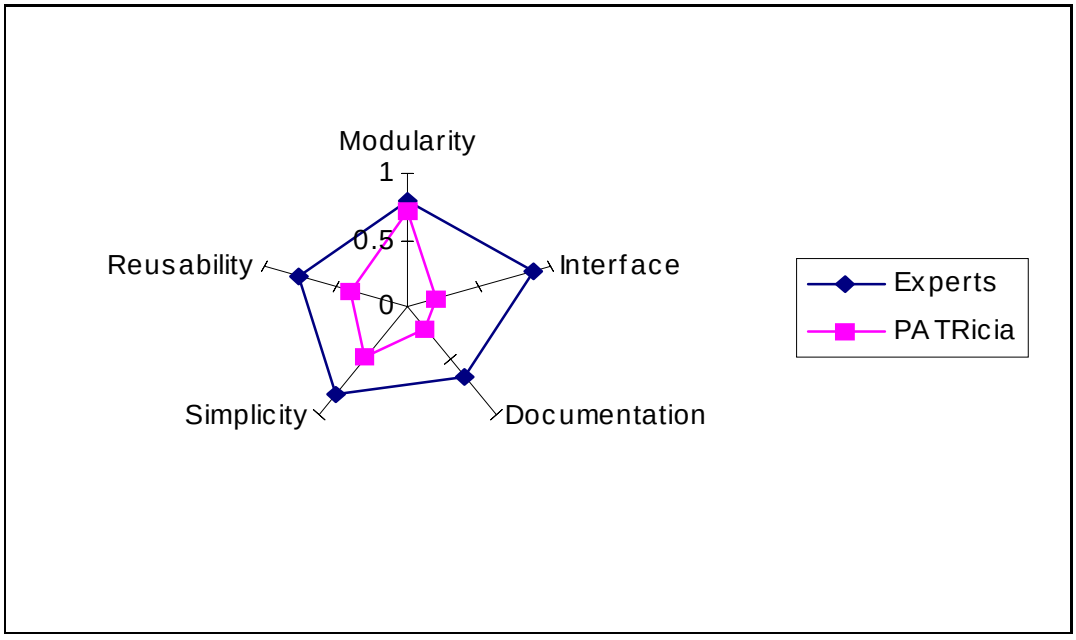


Figure 70. wxMouseEvent

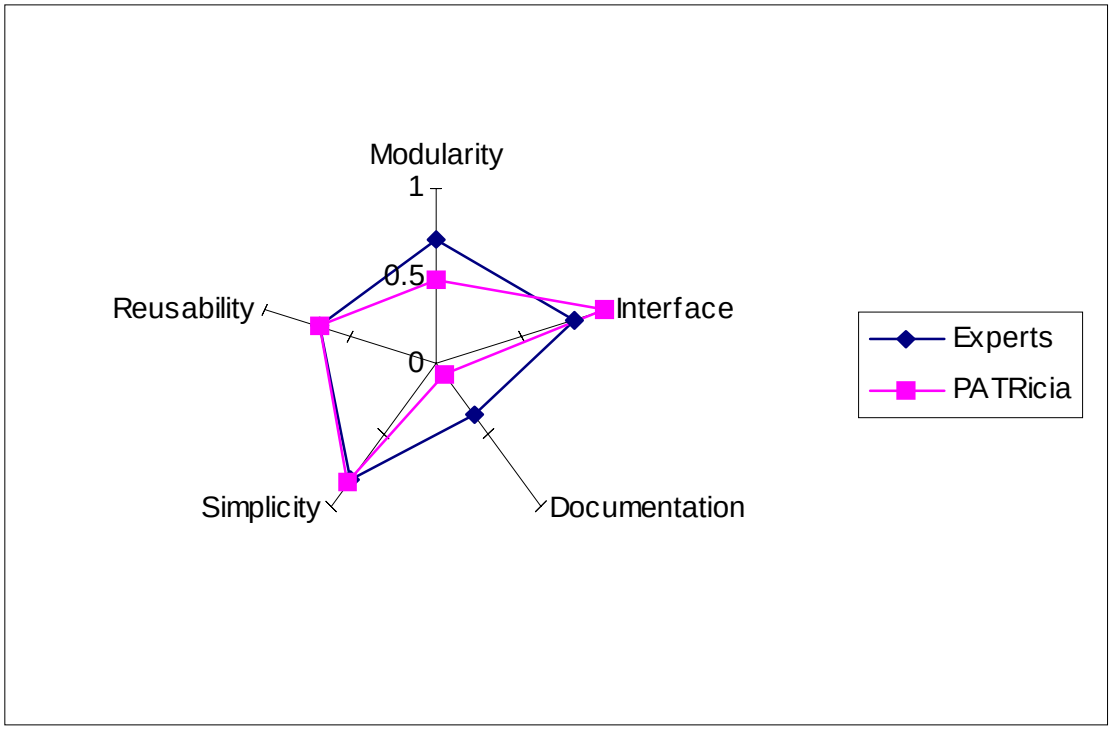


Figure 71. GINA Hierarchy

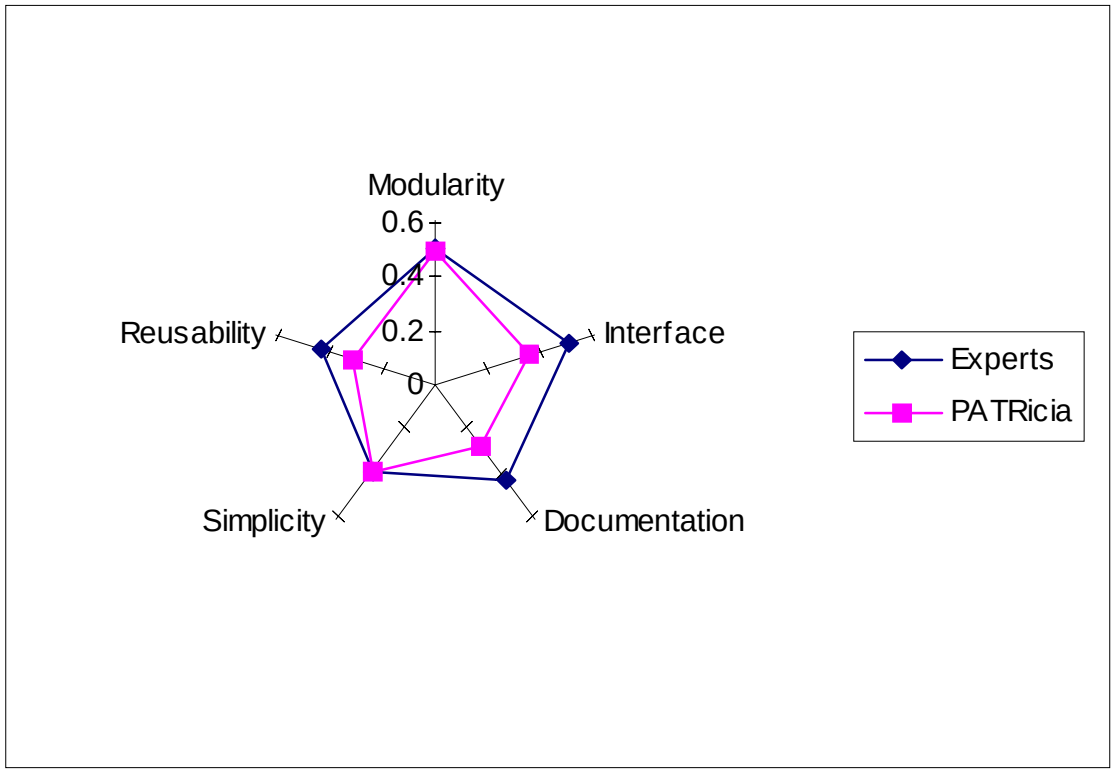


Figure 72. Watson Hierarchy

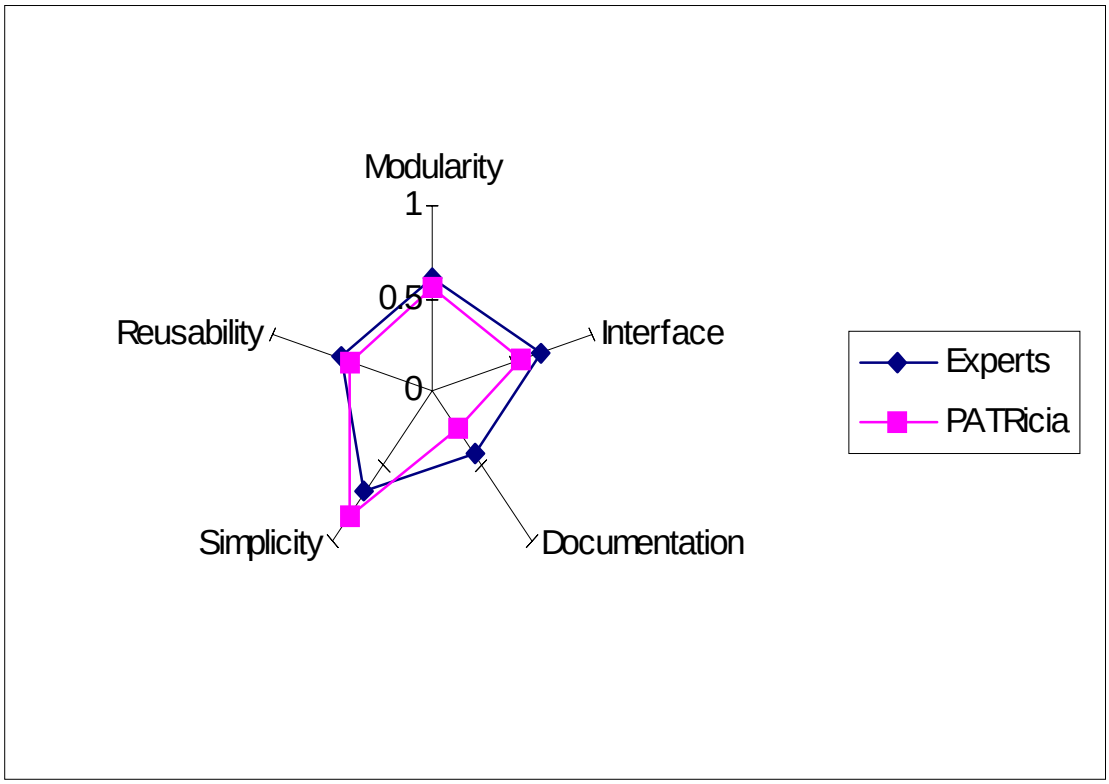


Figure 73. wxWindows Hierarchy #1

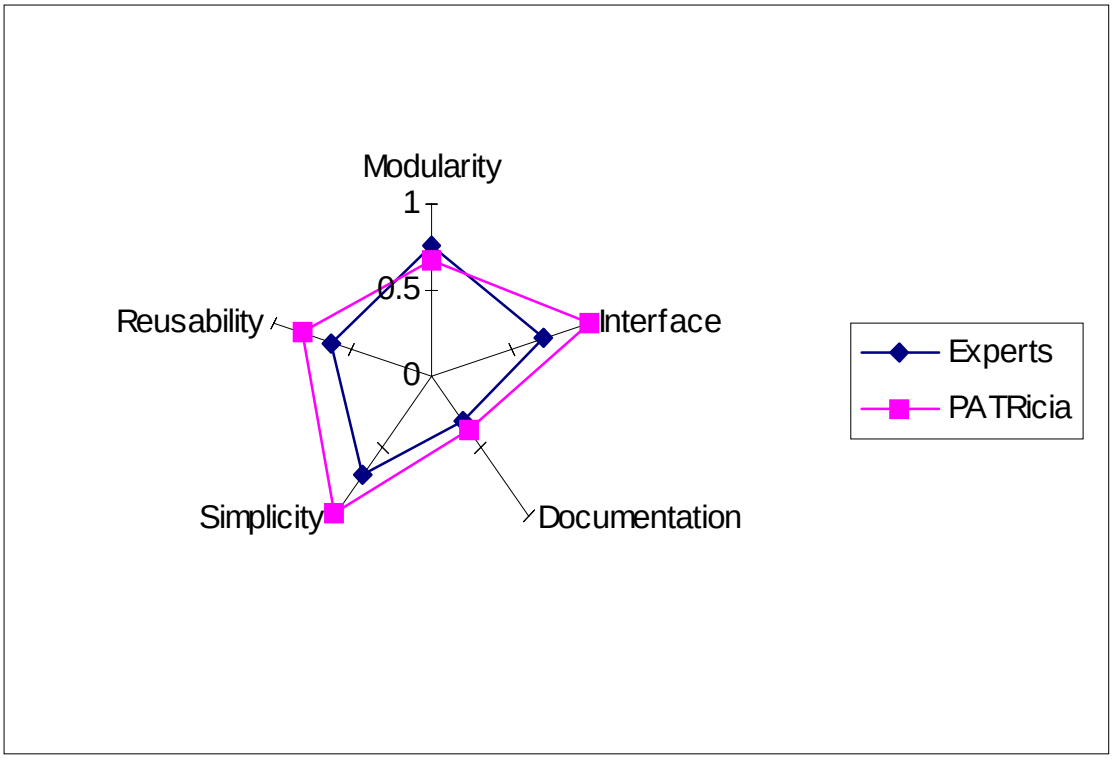


Figure 74. wxWindows Hierarchy #2

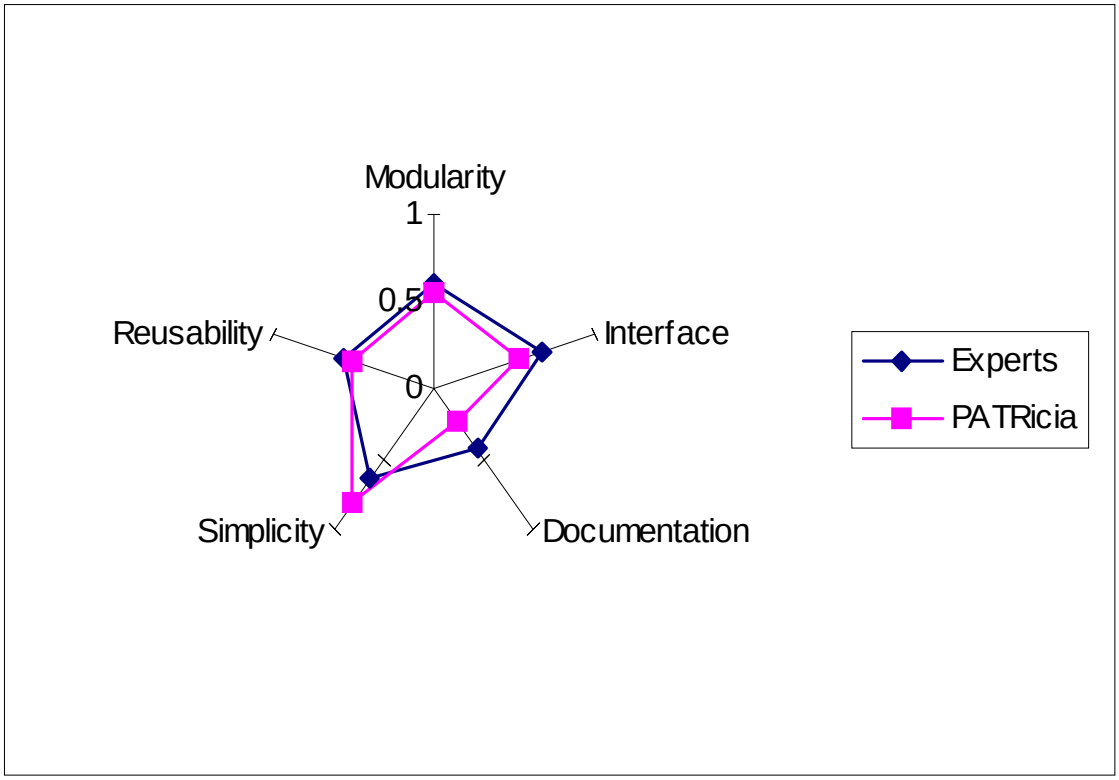


Figure 75. wxWindows Hierarchy #3

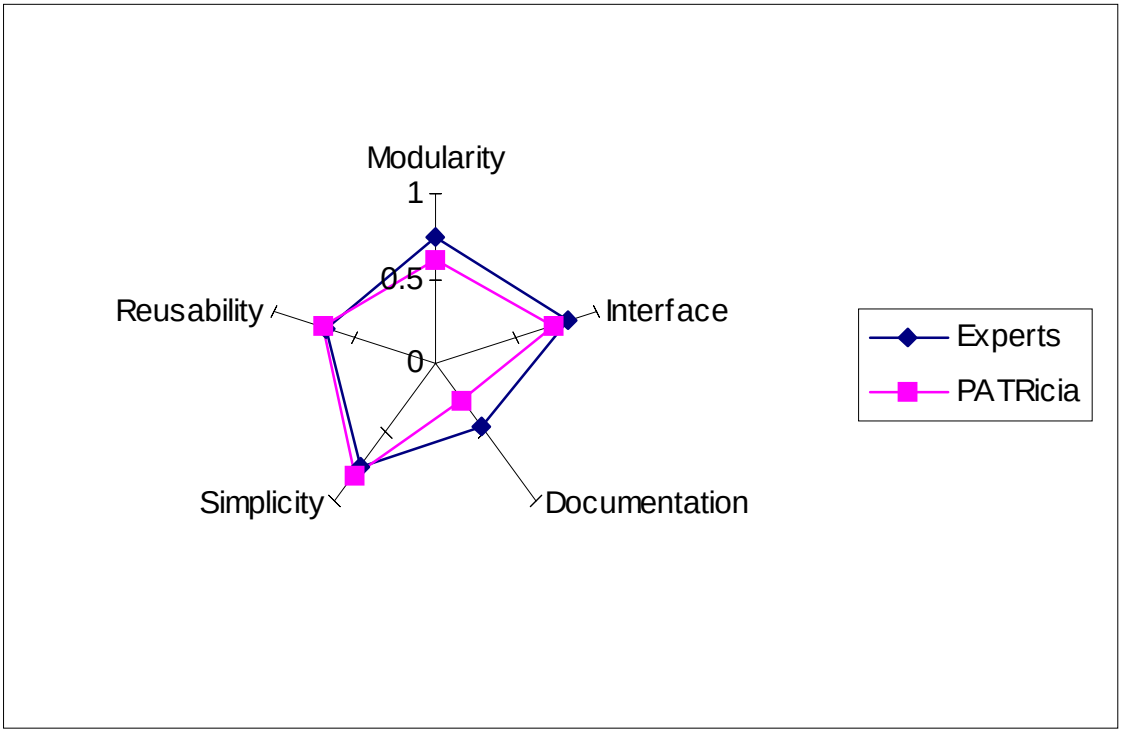


Figure 76. wxWindows Hierarchy #4

Chapter VIII

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The identification of reusable components in object-oriented software can be automated to a high degree. Through careful attention to knowledge extraction, knowledge base development, and metrics analysis, tools can be built that will present sufficient information to the user that an intelligent reuse choice can be made. The assumption that comments can provide sufficient information for determining what an object-oriented software component is doing has proven valid. While many other domains are yet to be explored, the domain used in this research was sufficiently complex to give a high degree of confidence in extendability to other domains. Building an effective knowledge-base to capture and analyze information about software products has proven key to the success of an automated software reusable components identification approach. This research has shown that the development of this knowledge-base is a difficult and tedious task and must be done for each domain of interest. In order to make understanding tools more effective, a better way to automate the capture of this information will have to be developed.

The gathering of information about potential reusable software components is quite doable. However, the metrics used to assess software reusability are currently very subjective and much work needs to be done to quantify and improve this assessment process.

While the conclusions of this research are very positive, many additional areas that need to be resolved have been identified. These can be broadly classified in the areas of tool improvements, domain expansion, improved artificial intelligence concepts, and better metrics.

When considering tool improvements, the initial version of the **PATR**icia system lacks several features that would facilitate its use in a production environment. The most important of these is

a graphical user interface. This would have the added advantage of allowing the tool to be used as a maintenance tool as well as a legacy software reuse tool. A related need would be a better explanation facility for the **PATR**icia system since the current explanation facility overwhelms with data. Also, the **PATR**icia system currently works only on C++ software and it would be useful to extend it to other computer languages. For use in other countries, the **PATR**icia system could also be extended to operate on comments in other natural languages than English.

There is a need to have a better way to categorize a component for insertion into a software reuse library. This might be done by using the concepts identified for a keyword list. This could be extended to implement some of the currently used faceted classification approaches.

When considering domain expansion, the **PATR**icia system could be extended to operate in other domains with the addition of a semantic net generation tool that would reduce the difficulty of creating a new knowledge-base. Also, the problem of domain shrinking was investigated, but not completely solved during this research. The knowledge-base of the **PATR**icia system could be re-implemented in a medium that allows for more control of the salience of objects such as CLIPS rules.

Several artificial intelligence-related investigation areas have been identified. First, the knowledge-based techniques available in the **PATR**icia system could be refined to identify and possibly quantify different programming styles. This could also be used as a teaching tool for students as well as an aid to standards enforcement. Next, the natural language processing portion of the **PATR**icia system could be extended to handle on-line documentation. While this becomes a much more general natural language processing problem, by applying the sublanguage concepts it is possible that computer documentation in general could be handled. Additionally, the **PATR**icia system could be extended to allow the analysis of software developed in the functional decomposition paradigm.

Several investigation areas related to metrics have been identified. There is a need to quantify how well a certain class' functionality matches the reuse area of interest. This might be

accomplished by a count of the number of concepts identified compared to the number of lines of code. This could be used to identify a key class or a non-key class according to the definitions by Lorenz and Kidd [78].

There is a continual need for better and more refined metrics. It became clear during the metrics portion of this project that "Granularity" is a better measure of Modularity than is Cohesion. If all classes are perfectly cohesive then many classes will be required to perform even a fairly simply task. This also tends to result in very deep class hierarchy trees, which are not desirable from a complexity standpoint. The application of knowledge-based techniques could potentially be used in a granularity calculation.

It would be very useful to find a way to measure the quality of identifier names in order to improve the calculation of the Interface and Documentation reusability quality factors in the **Metrics Analyzer**. Possibly a knowledge-based approach with some heuristics would help in this area.

A related problem that needs to be explored is the need to quantify how well a class is documented from a semantic, understanding standpoint, rather than just counting comments. This could be used during development, or as a teaching tool for students. It could also be used in the **Metrics Analyzer** in the calculation of the Documentation reusability quality factor.

APPENDIX A

STUDY OF COMMENTS AS A SUBLANGUAGE

Identified Types of Comments:

I) Sentence form, present tense:

a) indicative mood, active voice. For example, "This routine reads the data."

b) indicative mood, active voice, missing subject. For example, "Reads the data."

c) imperative mood, active voice. For example, "Read the data."

d) indicative mood, passive voice. For example, "This is done by reading the data."

e) indicative mood, passive voice, missing subject.

For example, "Is done by reading the data."

II) When not in sentence form:

A) Definition format

a) Itemname -- definition. For example, "MaxLength -- Maximum CFG Depth"

b) definition. For example, "Maximum CFG Depth"

Most of this type of comment are associated with variable definitions. Another category of this type of comment consists of the comments that signal areas within a file. For example, "Start of Get Area"

B) unattached prepositional phrase.

For example, "used for application data", or "to support scrolling text".

C) value definitions. For example, "0 = not selected, 1 = is selected"

D) Mathematical formulas. Can be Boolean expressions, in some cases.

Note that mathematically intensive code would normally have more of this type of comment.

III) Content

a) operational description.

b) definition

c) description of definition. For example, "This defines a NIL value for a list."

d) instructions to reader. For example, "See the header at the top of this file."

IV) simple future tense, indicative mood

V) simple past tense, indicative mood

For the following comment analyses, a comment stripper program was used to extract the comments from the code. The order of the comments was chosen by performing an ls *.* >mm command in Linux in each directory containing software of interest; the comment stripper then accessed the files in order as they appeared in the mm file, and placed the resulting comments in a single comment file. The first 108 comments in the comment file were examined. This process achieves a reasonable amount of arbitrary selection/randomness, since the 108 comments typically includes comments from several files within a particular package, and especially since comments such as revision notices and copyright notices are ignored (the revision and copyright comments are considered to be non-representative of the operation of the software, and therefore not of interest to this study). Similarly, comments consisting of commented-out code were ignored. The reasoning here is that in this case comment characters are performing two duties -- one duty is to document the software (the interest area of this project), and the other duty (as shown by commented-out code) is to (presumably temporarily) remove code from the compilation process. In one case (the wxWindows case), an exceptionally large file consisting solely of data definitions (called common.h) was skipped since it was felt that this file erroneously skewed the number of definition-style comments vs. sentence-style comments when compared to the software as a whole, when considering only the first 108 comments.

Analysis of Ga Tech C++ Parallelization Tool*

Format | | V Content ---->		IIIa)	IIIb)	IIIc)	IIId)	other
40.7%	I a)	37 (34.3%)	0	7 (6.48%)	0	0
4.6%	I b)	5 (4.6%)	0	0	0	0
23.2%	I c)	24 (22.2%)	0	0	1 (0.9%)	0
0.9%	I d)	1 (0.9%)	0	0	0	0
0%	I e)	0	0	0	0	0
0%	II A) a)	0	0	0	0	0
11.1%	II A) b)	0	12 (11.1%)	0	0	0
0%	II B)	0	0	0	0	0
0%	II C)	0	0	0	0	0
11.1%	IV)	12 (11.1%)	0	0	0	0
5.6%	V)	6 (5.6%)	0	0	0	0
2.8%	other	0	0	0	0	3(2.8%)

* Note: Comments dating code revisions, and listing the person performing the revision were ignored.

Total number of sentence style comments = (37+7+5+24+1+1) + (12) + (6) +(3) = 96

Total number of sentence style comments in present tense = (37+7+5+24+1+1) = 75

Total number of sentence style comments not in present tense = 12+6+3 = 21

%sentence style comments in present tense = 75/96 = 78%

%sentence style comments not in present tense = 21/96 = 22%

Analysis of Watson DOS GUI Package*

Format
|
|
V Content ---->

		IIIa)	IIIb)	IIIc)	IIId)	other
36.1%	I a)	39 (36.1%)	0	0	0	0
0%	I b)	0	0	0	0	0
3.7%	I c)	3 (2.8%)	0	1 (0.9%)	0	0
0%	I d)	0	0	0	0	0
0%	I e)	0	0	0	0	0
0%	II A) a)	0	0	0	0	0
37.0%	II A) b)	0	40 (37.0%)	0	0	0
6.5%	II B)	7 (6.5%)	0	0	0	0
10.2%	II C)	11 (10.2%)	0	0	0	0
3.7%	IV)	4(3.7%)	0	0	0	0
0%	V)	0	0	0	0	0
2.8%	other**	0	0	0	0	3 (2.8%)

* Note: Comments containing copyright notices were ignored.

** Note: Comments containing code that for some reason had been commented out were ignored.

Total number of sentence style comments = (39+3+1) + (4) + (3) = 50

Total number of sentence style comments in present tense = (39+3+1) = 43

Total number of sentence style comments not in present tense = 4 + 3 = 7

%sentence style comments in present tense = 43/50 = 86%

%sentence style comments not in present tense = 7/50 = 14%

Analysis of wxWindows GUI package

Format | | V Content ---->		IIIa)	IIIb)	IIIc)	IIId)	other
13.9%	I a)	15 (13.9%)	0	0	0	0
8.3%	I b)	8 (7.4%)	1 (0.9%)	0	0	0
13.0%	I c)	14 (13.0%)	0	0	0	0
0.9%	I d)	1 (0.9%)	0	0	0	0
0%	I e)	0	0	0	0	0
0%	II A) a)	0	0	0	0	0
61.1%	II A) b)	0	66 (61.1%)	0	0	0
0.9%	II B)	1 (0.9%)	0	0	0	0
0%	II C)	0	0	0	0	0
0.9%	IV	1 (0.9%)	0	0	0	0
0.9%	V	1 (0.9%)	0	0	0	0
0%	other	0	0	0	0	0

Total number of sentence style comments = (15+8+1+14+1) + (1) + (1) = 41

Total number of sentence style comments in present tense = (15+8+1+14+1) = 39

Total number of sentence style comments not in present tense = 1+1 = 2

%sentence style comments in present tense = 39/41 = 95%

%sentence style comments not in present tense = 2/41 = 5%

Analysis of GINA GUI Package*

Format
|
|
V Content ---->

		IIIa)	IIIb)	IIIc)	IIId)	other
22.2%	I a)	22 (20.4%)	2 (1.8%)	0	0	0
3.7%	I b)	1 (0.9%)	3 (2.8%)	0	0	0
18.5%	I c)	2 (1.9%)	18 (16.7%)	0	0	0
0%	I d)	0	0	0	0	0
0%	I e)	0	0	0	0	0
0%	II A) a)	0	0	0	0	0
44.4%	II A) b)	0	48(44.4%)	0	0	0
0%	II B)	0	0	0	0	0
0%	II C)	0	0	0	0	0
1.9%	IV	2 (1.9%)	0	0	0	0
1.9%	V	2 (1.9%)	0	0	0	0
7.4%	other	0	0	0	0	8 (7.4%)

* Note: Some GNU code and comments were included as part of the GINA package. Note that many files in GINA have no comments at all, other than filename, last edit, etc. (those comments were not included in this study). It required many GINA files to achieve 108 comments.

Total number of sentence style comments = (22+2+1+3+2+18) +(2)+(2)+(8) = 60

Total number of sentence style comments in present tense = (22+2+1+3+2+18) = 48

Total number of sentence style comments not in present tense = 2+2+8 = 12

%sentence style comments in present tense = 48/60 = 80%

%sentence style comments not in present tense = 12/60 = 20%

Overall

Format V		**Content ----> IIIa)**	**IIIb)**	**IIIc)**	**IIId)**	**other**
28.24% 122	**I a)**	**113(26.16%)**	**2 (0.46%)**	**7 (1.62%)**		
4.17% 18	**I b)**	**14(3.24%)**	**4 (0.93%)**			
14.58% 63	**I c)**	**43(9.95%)**	**18 (4.17%)**	**1(0.23%)**	**1(0.23%)**	
0.46% 2	**I d)**	**2(0.46%)**				
	I e)					
	II A) a)					
38.4% 166	**II A) b)**		**166(38.43%)**			
1.85% 8	**II B)**	**8(1.85%)**				
2.55% 11	**II C)**	**11(2.55%)**				
4.40% 19	**IV**	**19(4.40%)**				
2.08% 9	**V**	**9(2.08%)**				
3.24% 14	**other**					**14(3.24%)**

Note: Total = 108 X 4 = 432 comments examined

Total number of sentence style comments = 96+50+41+60=247

Total number of sentence style comments in present tense = 75+43+39+48=205

Total number of sentence style comments not in present tense = 21+7+2+12=42

%sentence style comments in present tense = 205/247 = 83%

%sentence style comments not in present tense = 42/247 = 17%

APPENDIX B

STUDY OF SYNTACTIC TAGGING OF IDENTIFIERS

Identifiers of variable names and function names from C++ class definitions were examined, and various common identifier formats were determined. The classes from which these identifiers were derived were chosen from three independent GUI implementations (wxWindows[102], Watson XWindows[111], and GINA[8]), from a parallelization tool [88](called PAT) developed at Georgia Tech, and from example classes in the book The Complete C++ Primer[112].

Formats

Notation: **adj** = either adjective or an adjectival (a word that occupies an adjective's position but does not show degrees of comparison)

verb, noun, preposition = defined as in English

Items in brackets [obj], [obj. prep] = describe the usage of the preceding noun

& = empty string

{ }* = 0 or more strings of the type specified in the set (i.e. empty string & is allowed)

Common Formats of variable identifiers:

Format 1: {adj}* noun

Format 2: {{adj}* noun }* preposition {adj}* noun [obj.prep]

Common Formats of function identifiers:

Format 3: verb {adj}* noun [obj.]

Format 4: verb {adj}* noun [obj] infinitive verb

Format 5: noun adverb (verb is assumed)

Format 6: {adj}* noun [obj] verb

Format 7: {not + &} verb

(can be single verb, combination of verbs such as "is shown")

Format 8: verb {noun + &} preposition {adj}* noun { {and {adj}* noun} + & }

Less common Formats of variable identifiers:

Format 3 occasionally is used for variables as well as functions

Less common Formats of function identifiers:

Format 1 occasionally is used for functions as well as variables

Note: Formats 1,2, 3, 7, and 8 represent English sentence portions (Formats 3 and 8 are parsable as complete sentences), and are in normal English grammatical order. Formats 5 and 6 are not in normal English grammatical order.

Summary for Each Class:

File wx_button.h, class wxButton (wxWindows GUI):

variables:

2 Format 1

functions:

6 Format 3

1 Format 7

File applib.h, class TAppWindow (Watson XWindows GUI):

variables:

4 Format 1

2 Format 2

functions:

1 Format 1

12 Format 3

2 Format 5

1 Format 6

2 Format 8

File GnMouseDownCommand.h, class GnMouseDownCommand (GINA GUI):

variables:

3 Format 1

2 Format 3

1 No Standard Format

functions:

4 Format 1

7 Format 3

1 Format 5

2 Format 6

1 Format 7

1 No Standard Format

The Complete C++ Primer, p. 54, class temp_dial:

variables:

4 Format 1

2 No Standard Format

functions:

2 Format 3

The Complete C++ Primer, p. 98, class nest_egg:

variables:

1 Format 1

functions:

4 Format 7

The Complete C++ Primer, p. 179, class stack:

variables:

2 Format 1

functions:

2 Format 7

The Complete C++ Primer, p. 181, class piggy_bank:

variables:

1 Format 1

functions:

2 Format 7

The Complete C++ Primer, p. 211, class button:

variables:

1 Format 1

functions:

1 Format 3

1 Format 7

The Complete C++ Primer, p. 212, class menu:

variables:

2 Format 1

1 No Standard Format

functions:

1 Format 3

File wx_menu.h, class wxMenu (wxWindows GUI):

variables:

4 Format 1

1 No Standard Format

functions:

8 Format 3

5 Format 7

File wx_Menu.h, class wxMenuBar (wxWindows GUI):

variables:

none

functions:

5 Format 3

3 Format 7

File wx_dialg.h, class wxDialogbox (wxWindows GUI):

variables:

5 Format 1

1 Format 3

functions:

5 Format 3

6 Format 7

File GnCommand.h, class GnCommand (GINA GUI):

variables:

2 Format 1

2 Format 3

functions:

1 Format 1

7 Format 3

3 Format 7

2 Format 8

6 No Standard Format

File datastruct.h, class Hast_Table (Ga Tech parallelization tool):

variables:

3 Format 1

functions:

1 Format 1

1 Format 7

File depend.h, class dependence (Ga Tech parallelization tool):

variables:

1 Format 1

functions:

3 Format 1

2 Format 7

1 No Standard Format

File depend.h, class Hole_Dependence (Ga Tech parallelization tool):

variables:

none

functions:

3	Format 3
4	Format 7

File flowg.h, class flownode (Ga Tech parallelization tool):

variables:

11	Format 1

functions:

1	Format 1
7	Format 3

Results of Identifier Study

This analysis was performed over 17 classes, from 5 independent sources.

variables:

78.0%	46	Format 1
3.4%	2	Format 2
8.5%	5	Format 3
1.7%	1	Format 8
8.5%	5	No Standard Format
	----	
	59 total	

functions:

8.6%	11	Format 1
50.0%	64	Format 3
2.3%	3	Format 5
2.3%	3	Format 6
27.3%	35	Format 7
3.1%	4	Format 8
6.3%	8	No Standard Format
	----	
	128 total	

Using the algorithm for variables described earlier (see Figure 16):

	91.6%	Format 1, Format 2, most Format 3 and Format 8 identifiers are handled.
Problem:	8.4%	All No Standard Format identifiers will be **erroneously** identified as Format 1.

Using the algorithm for functions described earlier (see Figure 15):

82.7% Format 3, Format 6, Format 7, Format 8 identifiers are handled.

17.3% Format 1, Format 5, and No Standard Format identifiers will be asserted with no usage information. (didn't help, but didn't hurt)

Note: In functions, all no standard format identifiers were in the Gina package, except for one found in the Georgia Tech parallelization tool package. Also, in variables, 4 of 5 Format 3 identifiers (uncommon for variables since contain a verb) were found in the Gina package. This can perhaps be understood since the Gina package originated in Germany, and was presumably written by non-native English speakers. This assumption goes along with the fact that the Gina package was the most sparsely commented of the three GUI packages examined.

Only the algorithm for function identifiers has currently been implemented. The reason that the function algorithm has been implemented whereas the variable algorithm has not yet been implemented, even though the variable algorithm has a higher success rate, is that the variable algorithm does result in some false identifications.

Using the function algorithm alone, the PATRicia system was run on some classes (the applib.h file, classes Application and TAppWindow) it had been tested on before, prior to the addition of the function algorithm. In class TAppWindow, there was a "plot_line" identifier and a "clear_display" identifier. Previously, "line", obviously a noun, usage subject here, had been tentatively identified in the keyword layer of the PATRicia system knowledge base as a noun, usage object as well as a noun, usage subject. Also previously, "clear" had been tentatively identified in the PATRicia system knowledge base as an adjective, as well as a verb. After the addition of the function algorithm, "line" was appropriately identified only as a noun, usage object, and clear was appropriately identified only as a verb. Therefore, the function algorithm can result in an appropriate word-sense identification, that can reduce the occurrence of erroneous conclusions.

Class Variables and Functions and Their Associated Formats

Description: **adjectives and adjectivals are italicized**

adverbs are both italicized and underlined

nouns and pronouns are bold

verbs are underlined

File wx_buttn.h, class *wx***Button**:

variables:

*buttonBit***Map**	Format 1
*wx***Button**	Format 1

functions:

Create	Format 3
Set**Size**	Format 3
Set**Default**	Format 3
Set**Label**	Format 3
Get**Label**	Format 3
Command	Format 7

(from the context, "command" is a verb here, such as "I command you to depart")

Change**Colour**	Format 3

(Note the British spelling of "color")

File applib.h: class *TApp***Window**:

variables:

*string*_**height**	Format 1
*string*_**width**	Format 1
*saved*_**text**	Format 1
number_of_*saved*_**lines**	Format 2

*current_ring_buffer*_**start**	Format 2
redraw_both_text_and_graphics	Format 8

functions:

init_*scrolling*_**text**	Format 2
put_*scrolling*_**text**	Format 3
reset_*scrolling*_**text**	Format 3
plot_**line**	Format 3
plot_**string**	Format 3
plot_**rect**	Format 3
erase_**rect**	Format 3
clear_**display**	Format 3
do_**edit**	Format 3 (here "edit" is a noun)
show_**info**	Format 3
choose_**one**	Format 3
choose_**one**_from_**list**	Format 8
choose_**file**_*to_read*	Format 4
choose_**file**_*to_write*	Format 4
mouse_*down*	Format 5
mouse_*up*	Format 5
mouse_move	Format 6
do_*menu*_**action**	Format 3
update_**display**	Format 3
*idle*_**proc**	Format 1

File GnMouseDownCommand.h, class *GnMouseDown***Command**

variables:

start_**x**	Format 3
start_**y**	Format 3
*last*_**x**	Format 1
*last*_**y**	Format 1
timer	Format 1
mouse_already_moved	No Standard Format

functions:

submit	Format 7
name	Format 1
constrain_**mouse**	Format 3
draw_**feedback**	Format 3
track_**mouse**	Format 3
mouse_idle	No Standard Format
not_submitted	Format 7
scroll_before_redo	No Standard Format
hysteresis	Format 1
*auto*_**scrolling**	Format 1
*idle*_**timeout**	Format 1
call_**doit**	Format 3
handle_**timeout**	Format 3
*idle*_**action**	Format 1
motion_notify	Format 6
button_release	Format 6
get_*last*_**x**	Format 3
get_*last*_**y**	Format 3

The Complete C++ Primer, p. 54, class *temp*_**dial**

variables:

low	No Standard Format
high	No Standard Format
*dial*_**increment**	Format 1
	(here "increment" is a noun)
*x***loc**	Format 1
*y***loc**	Format 1
diameter	Format 1

functions:

display_**temp**	Format 3
set_*temp*_**range**	Format 3

The Complete C++ Primer, p. 98, class *nest*_**egg**

variables:

savings	Format 1

functions:

open	Format 7
deposit	Format 7
withdraw	Format 7
balance	Format 7

The Complete C++ Primer, p. 179, class **stack**

variables:

data	Format 1
top	Format 1

functions:

push	Format 7
pop	Format 7

The Complete C++ Primer, p. 181, class *piggy*_**bank**

variables:

savings	Format 1

functions:

deposit	Format 7
withdraw	Format 7

The Complete C++ Primer, p. 211, class **button**

variables:

label	Format 1

functions:

display_**label**	Format 3
activate	Format 7

The Complete C++ Primer, p.212, class **menu**

variables:

entries	Format 1
num_items	No Standard Format
*current*_**item**	Format 1

functions:

display_**entries**	Format 3

File wx_menu.h, class *wx***Menu**

variables:

num_Columns	No Standard Format
*button***Widget**	Format 1
*menu***id**	Format 1
*top*_**menu**	Format 1
*panel***Item**	Format 1

functions:

Append**Separator**	Format 3
Append	Format 7
Enable	Format 7
Check	Format 7
Checked	Format 7
Set**Title**	Format 3
Get**Title**	Format 3
Set**Label**	Format 3
Get**Label**	Format 3
Break	Format 7
Create**Menu**	Format 3
Destroy**Menu**	Format 3
Find*Menu***Item**	Format 3

File wx_menu.h, class *wxMenu***Bar**

variables:

none

functions:

Enable	Format 7
Enable**Top**	Format 3
Check	Format 7
Checked	Format 7
Set**Label**	Format 3
Get**Label**	Format 3
Set*Label***Top**	Format 3
Get*Label***Top**	Format 3

File wx_dialg.h, class *wxDialog***Box**

variables:

*dialog***Title**	Format 1
*modal_***showing**	Format 1
*invisible***Resize**	Format 1
*dialog***Shell**	Format 1
PostDestroyChildren	No Standard Format
frame	Format 1

functions:

Create	Format 7
Set**Size**	Format 3
Set*Client***Size**	Format 3
Get**Position**	Format 3
Show	Format 7
IsShown	Format 7
Iconize	Format 7
Fit	Format 7
Set**Title**	Format 3
Get**Title**	Format 3

File GnCommand.h, class *Gn***Command**

variables:

document	Format 1
view	Format 1
	(here "view" is a noun)
view_*x*_**offset**	Format 3
view_*y*_**offset**	Format 3

functions:

name	Format 7
	(here "name" is a verb)
submit	Format 7
IsStorable	No Standard Format
Write_to_**Stream**	Format 8
Read_from_**Stream**	Format 8
executable	No Standard Format
undoable	No StandardFormat
*clock*_**cursor**	Format 1
causes_**change**	Format 3
do**it**	Format 3
redo**it**	Format 3
undo**it**	Format 3
commit	Format 7
scroll_before_redo	No Standard Format
scroll_before_undo	Not Standard Format
submit_to_**framework**	Format 3
execute_**undoit**	Format 3
execute_**redoit**	Format 3
undoit_before	No Standard Format
redoit_before	No Standard Format

File datastruct.h, class Hash table:

variables:

*our*_**table**	Format 1
*max*_**size**	Format 1
*empty*_**buckets**	Format 1

functions:

hash	Format 7
*remaining*_**spaces**	Format 1

File depend.h, class dependence:

variables:

ref	Format 1

functions:

key	Format 1
compare	Format 7
read_write	No Standard Format
*dep*_**ref**	Format 1
*dep*_**name**	Format 1
print	Format 7

File depend.h, class Hole_Dependence:

variables:

none

functions:

print	Format 7
expand	Format 7
expanded	Format 7
patch	Format 7
store_*pred*_**hole**	Format 3
store_*potential*_**write**	Format 3
store_*incoming*_**refs**	Format 3

File flowg.h, class flownode:

variables:

name	Format 1
op	Format 1
param	Format 1
param2	Format 1
linenumber	Format 1
exprs	Format 1
scope	Format 1
*name*_**table**	Format 1
*lookup*_**label**	Format 1
*dummy_node*_**count**	Format 1
*call_node*_**count**	Format 1

functions:

*Enclosing***Function**	Format 1
lookup_**label**	Format 3
enter_**label**	Format 3
find_**flownode**	Format 3
find_**lab**	Format 3
make_**callnode**	Format 3
make_**dummynode**	Format 3
replace_**name**	Format 3

APPENDIX C

STUDY OF THE APPLICATION OF THE SLEATOR AND TEMPERLEY PARSER TO COMMENTS

Comments from wxWindows:

Comment	#Parses
1) WxTimer provides simple timer functions.	1
2) Start the timer. ("the" added)	1
3) Stop the timer. ("the" added)	1
4) Every time a callback happens, these are set to point to the right values for drawing calls to work.	312
1st parse fails but is fairly close (11th parse works)	
First linkage: Correctly identified callback, values, calls as nouns. Correctly identified set and to point as verbs. Erroneously identified "for drawing calls to work", drawing as verb, calls as noun object, work as noun.	
5) Create a DC corresponding to a canvas.	3
1st parse is correct	
6) Most transactions involve a topic name and an item name.	1
7) Note that this limits the server in the ways it can send data to the client.	58
(21 parses with no P.P. violations)	
1st parse fails but is very close "in the ways it can send data to the client" modifies "server" instead of "limits" (9th parse is correct)	
8) This declares menus and menu bars.	1

Comment	#Parses
9) Get the horizontal spacing.	1
10) Start a new line.	1
11) Append to the end of the list. ("the" added)	1
12) Delete all nodes.	1
13) An event class represents a C++ class defining a range of similar event types; examples are canvas events, panel item command events. nouns = class[obj], class [obj], range [obj.], types [obj. of prep], examples[subj.], events [obj.]	2 1st parse is correct
14) Read the event's arguments from any input stream. verb = read nouns = event [possessive], arguments [obj] error - "from any input stream" modifies "arguments" instead of "read"	4 1st parse fails but is very close (2nd parse was correct)
15) Add a primary event handler, the normal event handler for this event. (replaced "-" with ",")	2 1st parse is correct
16) Register a new event class, giving the new event class type, its superclass, a function for creating a new event object of this class, and an optional description.	60 1st parse is correct
17) Register the name of the event.	1
18) Get a secondary wxPanel.	1
19) Find the letter corresponding to the mnemonic, for Motif.	14 1st parse is correct
20) After this is called, the metafile cannot be used for anything since it is now owned by the clipboard.	10 1st parse is correct

Comment **#Parses**

Comments from Watson Xwindows GUI

1) Need to set the pop-up menu titles. 1

2) We only allow either text or graphics to be active in a document window. 5
1st parse is correct

nouns = text[obj], graphics [obj], window [obj. prep]
verbs = allow
adverbial phrase = to be active ("active" is counted as an adj)
adjectives = document

3) The following flag is non-zero if we are using the document window for scrolling text. 12
1st parse fails but is very close
(5th parse was correct)

nouns = flag[obj], window [obj], text [obj. prep]
verbs = is, using
adjectives = following, document, scrolling
Erroneously defined "for scrolling text" as modifying "window" instead of "are using"

4) Note that this only looks good if the application program is careful not to draw in the scrolling text area. 4
1st parse is correct

5) Make room for the menu button. 3
1st parse is correct

6) List selection. 1

7) We need to give back this storage. 1

8) Draw this once to clear the previous line. 2
1st parse fails but is very close

verbs = draw, clear
nouns = line
adjectives = previous
Erroneously has "once" modifying "to clear"

9) Allocate storage for saved text lines. 1

Comment	#Parses
10) Leave room for future code modifications.	4
	1st parse is correct

Comments from Watson Macintosh GUI

11) Handle menu actions.	1
12) Increase this for large windows.	1
13) This is a ring buffer.	1
14) This can be either 16 or 32 bits.	2
(both linkages are the same)	1st parse is correct
15) The user accepts all responsibility for its use.	2
	1st parse is correct
16) Expand the heap so new code segments load at the top.	5
	1st parse is correct
17) Make a single document window for this application.	6
"single" listed as part of noun phrase instead of as an adjective	1st parse fails but is very close
	(2nd parse is correct)
18) Handle events.	1
19) Do idle work.	2
	1st parse fails, but is fairly close
	(2nd parse is correct)
"idle" is part of verb phrase "Do idle" rather than an adj. modifying work.	
20) Set the prompt text field.	2
	1st parse is correct

Comment **#Parses**

Comments from GINA

1) This is necessary to force initialization of global symbols that are contained in the shared library. 7

All parses fail

However, outcome is not too bad.

nouns = symbols [obj. prep], library [obj. prep], force [obj.prep]
verbs = initialization, contained
adjectives = global, shared
Erroneously, "force" is identified as a noun in all parses.

2) Define metaclass information and downcast operators. 2

1st parse is correct

3) This may look like C code, but it is really C++. 8

1st parse is correct

Note: GINA has very few comments--possibly because it was apparently written by non-native English speakers (apparently Germans) not comfortable in English--there were also a few German comments.

Comments from code in The Complete C++ Primer

1) Create a string object. 1

2) Read a new string. 1

3) Call the friend function. (both linkages are the same) 2

1st parse is correct

4) Copy null terminated string. 1

5) Copy a bound character array. 3

"bound" is part of a noun phrase instead of an adjective 1st parse fails but is very close (2nd parse is correct)

6) Call a virtual function. (both linkages are the same) 2

1st parse is correct

Comment	#Parses
7) Define a general kind of button and its handler.	1
8) Do default action here.	2
1st parse fails but is very close nouns = action verbs = do adjectives = default Erroneously, "here" modifies "action" rather than "do"	
9) Derive different types of buttons and their handlers.	3
1st parse is correct	
10) Popup code goes here.	1
11) Code for scrolling goes here.	1
12) Set up buttons here.	4
1st parse fails (2nd parse is correct) nouns = buttons [obj. prep] verbs = set Erroneously, "up" treated as a preposition, "here" modifies "buttons"	
13) Other class members go here.	1
14) The dial is now a class.	1
15) Private members go here.	2
"private" is part of a noun phrase instead of an adjective 1st parse fails but is very close	
16) Protected members go here.	1
17) Public members go here.	2
"Public" is part of a noun phrase instead of an adjective 1st parse fails but is very close	
18) Declare the object.	1
19) Define the object's size.	1
20) Calculate the object's area.	1

Comment	#Parses

Comments from the Georgia Tech Parallelization Tool

1) (This) returns the node on the far left in the subtree pathings.	5
"in the subtree pathings" modifies "left" instead of "node"	1st parse fails but is very close
2) (This) returns the node on the far right in the subtree pathings.	5
"in the subtree pathings" modifies "right" instead of "node"	1st parse fails but is very close
3) A list may be iterated over in one of two styles.	1
4) Back up the iterator by 1.	2
	1st parse is correct
5) Advance the iterator by 1.	2
	1st parse is correct
6) Return the current element.	2
	1st parse is correct
7) This file defines a wrapper class which allows the interpretation of the operator ()() member function of list iterators to mean 1 of 2 things depending upon context.	58 1st parse is correct
8) (This) allows one to tell if the iterator is empty.	3
	1st parse is correct
9) (This) returns the current value.	2
	1st parse is correct
10) Note that we only allow graph iterators to advance.	5
"to advance" is treated as a prepositional phrase modifying iterators, whereas it should be an infinitive verb with allow.	1st parse fails but is fairly close
Note: removal of "advance" as a noun from the dictionary, probably not appropriate for software anyway, would fix the problem.	
11) This has the side effect of advancing the iterator past the element deleted.	25 (15 with no P.P. violations) 1st parse fails but is very close
"past the element deleted" modifies "iterator" instead of "advancing"	

Comment	#Parses
12) Advance the iterator	1
13) Return the current element in the iterator and advance. "current" treated as part of a noun phrase instead of as an adjective	17 1st parse fails but is very close
14) This allows one to tell if the iterator is empty.	3 1st parse is correct
15) The template takes 2 formal parameters.	1
16) The template functions assume these members exist in any class instantiated.	1
17) We shifted as much work as possible into the abstract class to reduce instantiation expense.	10 (4 with no P.P. violations) 1st parse is correct
18) K is the key used for ordering the list.	3 1st parse is correct
19) See the header at the beginning of this file.	5 1st parse is correct
20) We allow duplicates, but they must be put to the right.	2 1st parse is correct

Results

The experiment examined 83 comments from 5 different sources -- wxWindows GUI package, Watson Xwindows GUI package[99], Watson Macintosh GUI package[99], GINA GUI package[99], a Georgia Tech Parallelization tool package (called PAT)[99], and C++ code from the book The Complete C++ Primer[99].

Some comments were not parsable because they were not in sentence format (for example, consisted totally of nouns). A small number of comments in sentence form were not parsable by the Sleator and Temperley natural language parser (approximately 5).

Note that only three of the comments in this study was not in present tense. (Comment #20, wxWindows, has one past tense, one present tense, Comment #3, GaTech Parallelization Tool is in future tense, Comment #17, GaTech Parallelization tool is in past tense.).

Results related to Ambiguity of a Parse

Percentage	Number	Description
42.17%	35	Only a single parse, no ambiguity
36.14%	30	Multiple parses, first parse was the correct parse
14.46%	12	First parse was incorrect, but missed in very minor ways (such as a single adverb modifying the wrong verb, a prepositional phrase that is attached to the wrong word, or an adjective treated as part of a noun phrase) that will not affect the PATRicia system as currently defined.)
4.82%	4	First parse was incorrect, but the effect on the PATRicia system is minor
2.41%	2	First parse was incorrect
	83 total	

Note: Randomness was achieved by using the first 20 parsable comments (GINA did not have that many), except for comments relating to copyright at the top of a file. Additional randomness was achieved by the fact that file order was determined by the order in which files were printed.

Discussion of Results

For 78.31% (42.17% + 36.14%) of the comments, the first parse was the correct parse.

For 92.77% (42.17% + 36.14% + 14.46%), the first parse provides useful information to the PATRicia system, without having erroneous information affect the PATRicia system (as the PATRicia system is currently defined).

For 4.82%, some very minor erroneous information is presented to the PATRicia system, but most of the information is correct.

A summary follows of the analysis of comments that were labelled above as being comments with minor parse errors that will not affect the PATRicia system as currently defined:

wxWindows comments

7) Note that this limits the server in the ways it can send data to the client.

Erroneous attachment of a prepositional phrase. "in the ways it can send data to the client" modifies "server" instead of "limits"

14) Read the event's arguments from any input stream.

Erroneous attachment of a prepositional phrase. "from any input stream" modifies "arguments" instead of "read"

Watson Xwindows GUI comments

3) The following flag is non-zero if we are using the document window for scrolling text.

Erroneous attachment of a prepositional phrase. "for scrolling text" modifies "window" instead of "are using".

8) Draw this once to clear the previous line.

Erroneous attachment of an adverb. "once" modifies "to clear" instead of "Draw".

Watson Xwindows Macintosh GUI comments

17) Make a single document window for this application.

Adjective erroneously treated as a noun in a noun phrase. "single" listed as part of a noun phrase instead of as an adjective.

Comments from code in The Complete C++ Primer

5) Copy a bound character array.
Adjective erroneously treated as a noun in a noun phrase. "bound" listed as part of a noun phrase instead of as an adjective.

15) Private members go here.

Adjective erroneously treated as a noun in a noun phrase. "Private" listed as part of a noun phrase instead of as an adjective.

17) Public members go here.

Adjective erroneously treated as a noun in a noun phrase. "Public" listed as part of a noun phrase instead of as an adjective.

GaTech Parallelization Tool comments

1) (This) returns the node on the far left in the subtree pathings.
"in the subtree pathings" modifies "left" instead of "node"

2) (This) returns the node on the far right in the subtree pathings.
"in the subtree pathings" modifies "right" instead of "node"

10) Note that we only allow graph iterators to advance.
"to advance" is treated as a prepositional phrase modifying iterators, whereas it should be an infinitive verb with allow.
Note: removal of "advance" as a noun from the dictionary, probably not appropriate for software anyway, would fix the problem.

11) This has the side effect of advancing the iterator past the
element deleted.
"past the element deleted" modifies "iterator" instead of "advancing"

13) Return the current element in the iterator and advance.
"current" treated as part of a noun phrase instead of as an adjective

APPENDIX D

BACKGROUND OF THE EXPERTS

Lisa L. Bowen received the BS degree in Computer Science in 1993 and the MS degree in Computer Science in 1996 from the University of Alabama in Huntsville, Huntsville, AL. She is now a Lecturer in Computer Science and a PhD student at the University of Alabama in Huntsville. From 1985 to Feb.1987 she worked as a Systems Analyst at Delta Research, Inc., Huntsville, AL, where she developed engagement simulations for the Strategic Defense Initiative. From Feb. to Oct. 1987 she worked as a Systems Manager for PaneLogic, Inc., Huntsville, AL, where she designed and developed database and invoicing systems. From Oct. 1987 to Feb. 1988 she was employed as a Software Systems Engineer at Engineering and Economics Research, Huntsville, AL, where she developed analytic methods for Theater Missile Defense target databases and performed targeting analysis for SDI applications. From Feb. to May 1988, she worked for Titan Systems, Huntsville, AL, as a Software Engineer, where she designed and developed computational tools for the SDI. From May 1988 to April 1991, she was employed as a Systems Manager at Kaman Sciences, Huntsville, Alabama, where she managed several computer systems, and designed and developed database systems. She worked as a Software Engineer from April 1991 to June 1992 at Techni-Core, Huntsville, AL, where she designed and developed Windows-based password and auto-logon software for several different systems. From August 1992 to May 1997, she was employed as a Software Engineer at Coleman Research Corporation, Huntsville,AL, where she was responsible for the development and integration for the THAAD launcher simulation for the THAAD Verification and Validation System. She also served as the leader of the Requirements Management Process Action Team, where she was responsible for defining, implementing, and improving the Requirements Management Process.

David B. Etzkorn received a Bachelors of Electrical Engineering with highest honors in 1980, and the MS degree in Electrical Engineering in 1988 from the Georgia Institute of Technology, Atlanta, GA. He is currently employed at Huntsville Microsystems, Inc., as a Senior Design Engineer. At Huntsville Microsystems he develops hardware and software for microprocessor emulators. From 1980 to 1981 he was employed as a Design Engineer at Scientific Atlanta, Norcross, GA, where he performed analog and digital design for cable television equipment. From 1981 to 1987 he was employed as a Senior Design Engineer at Lanier Business Products, Atlanta, Georgia, where he performed hardware, firmware, and software design on microprocessor-based dictation equipment. From 1989 to 1992 he did research on personnel detectors as a Senior Design Engineer at Physitron, Inc., Huntsville, AL. His work at Physitron involved digital design, software design, and analog design.

LeeAnne W. Lewis received the BS degree in Business Information Systems and Quantitative Analysis in 1978 from Mississippi State University. She received the Master of Combined Science in Mathematics in 1986 from Mississippi College. She received the MS in Computer Science in 1993 from the University of Alabama in Huntsville, Huntsville, AL. She is currently employed as a Software Quality Manager at Ryder Transportation Systems, Miami, FL. At Ryder, she is responsible for developing software testing strategies, processes, plans, and environments for a variety of applications on client/server and multiple tiered architectures. From 1979 to 1994, she was co-owner of Clifton Lewis Trucking, Brandon, MS, where she had complete responsibility for the financial aspect of the business and developed database applications for a fuel analysis and reporting system for a fleet of tractor-trailer units. From 1982 to 1990 she worked as an information systems consultant for various non-profit organizations, and also taught mathematics and computer science courses at Hinds Community College, Jackson, MS. From 1985 to 1987 she worked as a Systems Analyst in the office of the Mississippi Secretary of State, Jackson, MS, where she performed system analysis and data normalization of large database systems, and was responsible for the acquisition of hardware and software. From 1990 to 1992 she was a Graduate Research Assistant at the University of Alabama in Huntsville, where she supported high performance computing for high school students and teachers on a state and national level. From 1992 to 1996 she was employed as a Senior Software Engineer at Quality Research, Huntsville, AL, where she served as a senior member of the Air Force SEI CMM Software Capability Evaluation team. She also served as a Program Manager responsible for the Life Cycle Software Engineering Support for the US Army Space and Strategic Defense Command Software Engineering division. She served on several verification/validation processes. She also developed requirements for a government executive information system, and for an event-based commercial software product.

Anthony Mark Orme received the BS in Computer Science in 1991 and will receive the MS in Computer Science in 1997 from the University of Alabama in Huntsville, Huntsville, AL. He is currently employed at Oracle Corp. as a Senior Instructor.
From 1988 to 1992 he was employed as a System/Database Administrator at the University of Alabama in Huntsville, where he designed, implemented, and supported various very large databases used in NASA-related research. He was in charge of daily maintenance and management of a VAX cluster, which included both Macintosh and PC clients connected via Pathworks. He was responsible for purchasing hardware and layered software to support research in the Optical Aeronomy Lab. From 1992 to 1995, he was employed as a Technical Lead/Software Development at CapMed Systems Corporation, Huntsville, AL, where he performed design and development of a client/server Clinical Patient Records System. This included choosing 4GL tools to handle the development and system lifecycle. He performed design and development of both the user interface and functional requirements of the project using Gupta Corp. SQLWindows and SQLBase. He developed regression testing procedures using Microsoft Test. He also

implemented and maintained a configuration control system. From Jan. 1996 to May 1997, he was employed as a Lecturer in Computer Science at the University of Alabama in Huntsville.

Bradley L. Vinz received the BS degree in Computer Science, Cum Laude, in 1987, and the MS degree in 1992 from the University of Alabama in Huntsville, Huntsville, AL. He currently is a PhD student in Computer Science at the University of Alabama in Huntsville, where his primary research area is the application of neural networks to image processing problems. From 1987 to 1998, he worked as an Associate Programmer Analyst at Unisys Corporation, Huntsville, AL, where he designed and implemented a user-interface to communicate with a complex network of computer modules. From 1988 to 1990, he was employed as a Software Analyst at Intergraph Corporation, Huntsville, AL, where he designed and implemented an interactive graphical user interface used in cartographic editing of digitized military defense maps.From 1990 to 1994 he was a NASA Fellow at the University of Alabama in Huntsville, where he performed research on neural networks. He designed and implemented several neural network simulators, all with user-friendly graphical interfaces. In 1994 he was the recipient of the Dynetics Corporation Endowed Scholarship. From 1994 to 1996 he was employed as a Lecturer in Computer Science at the University of Alabama in Huntsville.

Minyoung Yun received the BS degree in Textile Engineering in 1980 from Jeon-Buk National University in Korea. She received the MS in Computer Science in 1989 and the PhD in Computer Science in 1993 from the University of Alabama in Huntsville, Huntsville, AL. She is currently employed as an Assistant Professor at Sungkyul University, Korea. From Jan. 1994 to Dec. 1994 she was employed as an Assistant Professor at Alabama A&M University, Normal, AL.

Janet C. Wolf received the BS in Information and Computer Science, with honors, in 1982 from the Georgia Institute of Technology, Atlanta, GA. She is currently employed as a Development Programmer at IBM Research Triangle Park, North Carolina, where she recently worked on prototype team that developed the BSD 4.3 Sockets-over-SNA -- an implementation that allows TCP/IP Socket API programs to use SNA transport with no software rewrite. From 1982 to to 1987 she worked as a Development Programmer at IBM's EDX Communications Facility in Palo Alto, CA, where she designed, coded, tested, and maintained communications software for the IBM Series/1 Computer. From 1987 to 1992 she worked on VTAM

development as a Development Programmer at IBM Research Triangle Park, North Carolina, where she worked on teams that designed and developed VTAM X.25 support on VM, and for the MVS platform.

Randall P. Wolf received the BA degree in Computer Science, Political Science, and Sociology at Birmingham-Southern College, Birmingham, AL, in 1980. He received the MS in Computer Science at the University of Alabama in Huntsville, Huntsville, AL, in 1993. He currently is a PhD student in Computer Science at the University of Alabama in Huntsville, where his primary research area is knowledge acquisition via the integration of personal constructs and conceptual graphs. From 1980 to 1994 he was a Software Developer/Contractor for several different contracts. These contracts included work for the Center for Microgravity and Materials Research on the image analysis of crystal growth, and pattern recognition and data visualization of residual acceleration in orbital laboratories. Other contracts included work for General Research Corporation on the requirements analysis of the front end of a CASE tool to support the Strategic Defense Initiative, and work for Analysts International Corporation on the creation of the test plan for the OSI transport layer for a new operating system for the IBM AS400 computer. From 1981 to 1992 he worked for Computer Sciences Corporation, where he participated as a member of the launch crew for the first Space Shuttle flight. He also coded and debugged data communications simulations of Spacelab experiments communications, and maintained and improved statistics programs used in the analysis of wind vector data from Space shuttle launches. From 1982 to 1984 he worked for John M. Cockerham and Associates, where he designed and implemented improvements to cost and risk programs associated with the research and development of weapons systems. He also served as a database administrator for the Foreign Military Sales Data Management and Tracking System. From 1991 to 1993 he was a Graduate Research Assistant working on NASA sponsored research concerning residual acceleration levels in orbital laboratories -- this utilized pattern recognition, database technology, data visualization, Fourier analysis, and expert systems. From 1994 to 1996 he was a Graduate Research Assistant working on database security analysis research -- this included formal language techniques, as well as artificial intelligence techniques and pattern recognition and database techniques.

APPENDIX E

PROGRAM UNDERSTANDING EXPERIMENT PHASE I -- DIRECTIONS AND QUESTIONNAIRES

Directions

The object of this experiment is to compare the knowledge of C++ and GUI experts to the knowledge-base of the **PATR**icia system. The purpose of your portion of this project is to provide your knowledge of C++ GUI software in a format that can be compared to the output of the **PATR**icia system. There are 5 C++ and GUI experts taking part in this project. All of you were carefully chosen for educational background, and for experience. All of you have several years of programming experience. All of you also have technical degrees--either in computer science or electrical engineering; indeed, most of you have master's degrees, or are very near to completing a master's degree. I deeply appreciate your consenting to work on this project.

I have attempted to organize this experiment in such as way as to cost the experts the least possible time. In the notebook provided to you, you will find portions of four C++ GUI packages, the wxWindows GUI package[102] the Watson Xwindows GUI package[111], the Watson DOS GUI package[111], and the GINA GUI package[8]. All these are either shareware packages, or, in the case of the Watson GUI packages, was printed in a book, Portable GUI Development with C++, by Mark Watson[111]. Each class for you to analyze has had its class definition highlighted. Additionally, at the front of this notebook you have been provided with a list in alphabetical order of the current high level concepts in the **PATR**icia system knowledge-base, and

a list of the current keyword interface layer, also in alphabetical order, of the **PATR**icia system knowledge base.

You have been provided with two different types of report sheets. The first type, labeled at the top "Concepts Already In the PATRicia System Knowledge Base," is to be used when you find concepts in the software that already exist in the knowledge base. The second type, labeled at the top "New Concepts Not Currently In the PATRicia System Knowledge Base," is to be used when you find new concepts that do not currently exist in the knowledge base. I have provided you with several copies of each kind of sheet. Please feel free to copy off more as needed, or contact me and I will send you more.

Stage 1 (this is the most important part of the study):

Read the class definition of each highlighted C++ class. Please use **only** the comments and identifier names associated with the class**. We are interested in GUI-related concepts ONLY. Non-GUI concepts such as programming concepts that are not GUI-related should be ignored**. As you identify each main concept, look in the list of **PATR**icia system concepts, and find that concept. Then write the concept, and the class name where it was found, on a sheet of the type labeled "Concepts Already in the PATRicia System Knowledge Base." Please try to keep concepts from a single class sequentially written on the sheet as much as possible. (If you disagree with the definition provided with a particular concept, please notify me in some way. If you wish, you can write your definition on the list of concepts and return a copy of it to me). The list of **PATR**icia system concepts is in alphabetical order, by concept name, to try to make this task easier. However, it is possible that you know the concept under a different name than that used in the **PATR**icia system. You may be able to find the concept under its other name by looking at the concept definitions ("facts") provided. If you discover a concept that you did not find in the **PATR**icia system concept list, please include that concept, along with the name of the class where it was found, on a separate sheet, of the type labeled "New Concepts Not Currently In the PATRicia System Knowledge Base." In such a case, please look in the **PATR**icia system's keyword list, and identify any keywords that should map into that concept. Please write your best

definition of each newly identified concept in the area provided on the sheet. If you think additional keywords should map into that concept besides those provided on the keyword list, please include those additional keywords--however, please identifiy each new keyword (that is, each keyword that is not currently in the **PATR**icia system) as new by placing "(new)" beside the keyword.

Stage 2:

You have been given, on a diskette, a copy of the GnMouseDownCommand class. This class is part of the GINA GUI package, and has been sparsely commented. Please comment this class in your normal commenting style.

Stage3 (do if you have the time):

Go back through each selected class, and look at the code of the member functions (methods) associated with that class. For any concepts that you identify in this manner, that you did not identify just from the class definition, please handle as you did earlier by writing the concept name and class name, etc., on the appropriate sheet.. Please add, beside each class name, in parentheses, the word "code," and the name of the member function where the concept was identified. For example, given a concept "mouse," and a member function named "reset_mouse," in a class Mouse_Handler:

mouse Mouse_Handler (code, reset_mouse)

In all cases, I did not provide you with all member functions. The member functions are provided for the Watson packages, and for the GINA packages. I have highlighted the member functions definitions in green for those packages. Note that most of the Watson XWindows package member functions should be there, but possibly not all. For the Watson DOS package, all member functions should be there. For the GINA package, all member functions should be there.

When you have finished, please send me the sheets on which you have written the identified concepts. I will of course pay for postage. Tell me how much and I will pay you back, if that's all right. I have not yet decided whether to have you mail back the notebooks, or whether to have you keep them until the next phase of the project. Possibly you have opinions on this, such as perhaps you do not wish to have to store the notebook. I will probably have to update them in some way for the second phase of the project. If you wish to mail it back to me, do so, and I will pay for postage. Otherwise, keep it for now--I may have you mail it back later.

Speaking of which, there will be a later phase of this project. This experiment is to validate the knowledge base of the **PATR**icia system--the next phase will be to validate some **PATR**icia system quality metrics. I currently anticipate that phase will take place sometime next spring.

If possible, I would prefer to get your reports back sometime in early January--the earlier the better, from my standpoint. However, I do understand that this is not at the top of your task list over the holidays.

Thank you so much for your efforts!

Feel free to call me if you have any questions or difficulties.

	A	B	C	D	E	F	G	H	I	J	K	L
1		**Concepts Already In the PATRicia System Knowledge Base**										
2												_
3	**Concept**			**Class**								
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												

	A	B	C	D	E	F	G	H	I	J	K	L
26												
27												
28												
29												
30												
31												
32												
33												
34												
35												
36												
37												
38												

	A	B	C	D	E	F	G	H	I	J	K	L
1		New Concepts Not Currently In the PATRicia System Knowledge Base										
2												
3	Concept Name		Definition		Class		Keywords					_
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												

	A	B	C	D	E	F	G	H	I	J	K	L
26												
27												
28												
29												
30												
31												
32												
33												
34												
35												
36												
37												
38												

APPENDIX F

PROGRAM UNDERSTANDING EXPERIMENT PHASE II -- DIRECTIONS AND QUESTIONNAIRES

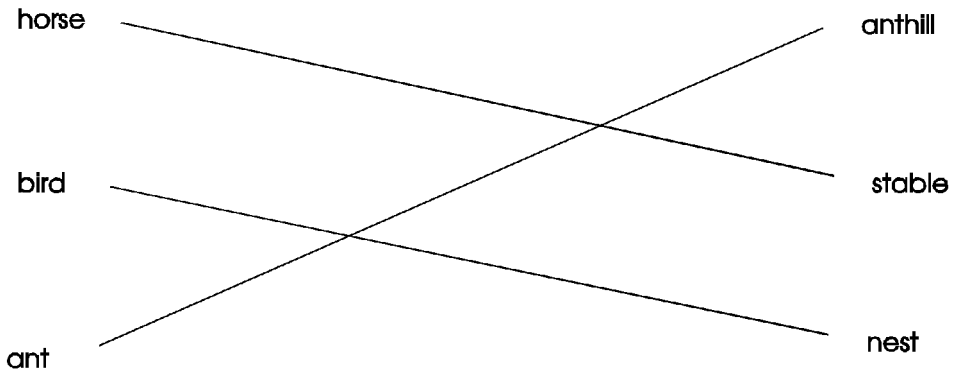
Directions

Hi again,

This should be a whole lot easier than last time. Don't be worried because there are a number of pages of stuff involved here--once you get going this should go very fast.(I think the hard part is understanding just exactly what I want you to do!)

The idea behind this experiment is to compare the concepts derived by the experts earlier (that means a merging of what you gave me last time, along with what the other experts gave me) with the output of the **PATR**icia system.

This is basically a simple matching process, like what you used to do in 3rd grade, for example:



Only in this case you're going to be matching expert-generated concepts to **PATR**icia system concepts.

As you recall, the domain of interest is that of graphical user interfaces (GUIs).

There are a few differences in the matching process here compared to what you used to do in third grade. Before I describe those differences, let me first give you a better idea of why this test is desired.

There are two aspects of the **PATR**icia system's output that we are interested in. First, we want to know how many of the concepts that were identified by the experts that the **PATR**icia system also identified (let's call this concept coverage). Second, for this experiment, we want to know how many **extra**, totally wrong concepts that the **PATR**icia system generates (if any) (let's call this concept overgeneration).

The idea, of course, is that there are two primary ways that the **PATR**icia system could be wrong. First, it might not identify all the concepts that it should. Second, it might identify lots of extra garbage that really isn't there at all.

Note that when we're talking about garbage, we primarily mean really wrong information, or concepts. For example, if you identify a window, you might imply other things normally associated with a window (such as erase window, redraw window, etc.) Even if those were not explicitly present in comments or identifiers in the class, it's not all all unreasonable to assume that somewhere associated with that class such items exist, and thus not totally wrong to identify them. However, some concepts might be totally wrong. For example, if you are working in a class that is identified as implementing Motif, and you identify it instead as being a Microsoft Windows class instead**. Please keep these distinctions in mind**.

For the reasons above, I've divided the matching process into two separate areas**. First**, two concepts might match exactly, or they might match as sub-concepts**. Second**, two concepts might be somewhat, or fairly related, even though not matching exactly.

An example of two concepts matching exactly would be to have a concept in the expert's concept list called "redraw_window," and you matched that to a concept in the **PATR**icia system's list called "redraw_window." (Note that on rare occasions these concepts might have different names, but be basically the same thing, or extremely closely related--such as "primary_window" and "workspace." In that case those concepts would be matched exactly). An example of matching sub-concepts would be to match the separate concepts "redraw" and "window" to "redraw_window." This is reasonable since those concepts have to be identified on the way to identifying "redraw_window."

An example of two concepts being related would be to have a concept in the expert's concept list called "move_mouse." Related concepts would include "move_mouse_down," "move_mouse_up," "mouse_events," and "mouse_event_messages." Matching these concepts is reasonable for the reasons described earlier.

The other possibility would be that the concepts did not really match at all. An example of this would possibly be "menu" and "start_timer," or, as we discussed before, "Motif" and "Microsoft_Windows."

Explicit Instructions, or HOW TO DO IT

You have been given two kinds of report sheets:

1) a report sheet listing, for each class, all the expert's concepts that were identified for that class. Note that each of the expert's concepts has a line reserved for you to write all **PATR**icia system concepts that are matched, or very nearly matched, or sub-concepts of the expert's concept-in-question. Note also that each of the expert's concepts has two lines reserved for you to write all **PATR**icia system concepts that are related, or somewhat related, to the expert's concept-in-question.

2) a report sheet, listing class names only, in which you are to write **PATR**icia system concepts that did not match, or be related to, other concepts (these concepts are extraneous garbage, or concept overgeneration--see above).

The report sheets are located in your notebook at the beginning of each package to be analyzed. The report sheets listing the expert's concepts are stapled together. They are followed by a single report sheet for listing the extraneous garbage.

In each package to be analyzed, following the report sheets, comes a page listing the class hierarchy that you are to follow in your analysis (you should start at the base class and go down). Then come the reports that were generated by the **PATR**icia system. There are basically three types of reports that are generated by the **PATR**icia system--a keyword report (that comes first in the notebook, following the class hierarchy sheet), a concept report (that follows the keyword report), and finally a class operational report (this should be last in the current package).

Summary of notebook order:

For each package to be analyzed,

1) report-sheets-for-matching, stapled together

2) garbage report sheet

3) class hierarchy sheet

4) keyword report

5) concept report

6) class operational report

For the purposes of this test, you are primarily interested in the concept report. On hopefully quite rare occasions you will look at the other reports--these rare occasions will be when there was neither a concept nor a related concept in the **PATR**icia system's concept report that matched the current-concept-being-examined from the expert's concept list.

You will probably want to **temporarily** remove your report sheets from the notebook while you match concepts**. Please write your matches, etc., on the report sheets. This will provide some needed uniformity for this test! Also, please replace them in the notebook afterwards.**

For each concept in a class, that concept might be listed as being inherited, or occurring in that particular class, or both. You can tell this by the "at location(s)" line**. Ignore inheritance while matching. Finally, however, when listing concepts on the garbage report, do not list inherited concepts** (they will have been handled already in the earlier class).

NOTE: inherited-only concepts will have their location marked in yellow!!!

Procedure:

1) Select a class to analyze.

2) Extract the report-sheets-for-matching (these are stapled together).

3) Turn to the **PATR**icia system concept report.

4) Look at each of the expert's concepts (for the current class) that is listed in the report-sheets-for-matching.

5) Looking at the **PATR**icia system concept report, find:

a) every concept that exactly matches the expert's concept-in-question. Remember that this may or may not have exactly the same name as the concept-in-question--more often it will. List this concept under the expert's concept-in-question on the report-sheet-for-matching, on the line labeled "Same or very similar concepts."

It is okay to use inherited concepts here!!!

Put a checkmark by each of the PATRicia system's concepts that you have used to match the expert's concept-in-question.

b) every concept that is a sub-concept of the expert's concept-in-question. For example, for the expert's concept-in-question "redraw_window," both "redraw" and "window" would be sub-concepts. List each sub-concept under the expert's concept-in-question on the report-sheet-for-matching, on the line labeled "Same or very similar concepts."

It is okay to use inherited concepts here!!!

Put a checkmark by each of the PATRicia system's concepts that you have used as a sub-concept to match the expert's concept-in-question.

c) every concept that is related, or somewhat related, to the expert's concept-in-question. For example, "scrolling_text" might be somewhat related to "copy-text_pointer," or "dialog_box" might be somewhat related to "working_dialog_box," or "message_dialog_box." List each related concept under the expert's concept-in-question on the report-sheet-for-matching, on the line labeled "Related concepts."

It is okay to use inherited concepts here!!!

Put a checkmark by each of the PATRicia system's concepts that you have marked as related to the expert's concept-in-question.

6) If no match or related concept is found for the expert's concept-in-question, then look at the **PATR**icia system's keyword report. If one or more keywords related to the expert's concept-in-question is found, then write "See keyword report." under the expert's concept-in-question on the report-sheet-for-matching, on the line labeled "Same or very similar concepts." **There is NO need to put a checkmark on the keyword report.**

7) If no match or related concept is found for the expert's concept-in-question, then look at the **PATR**icia system's class operational report. Read this report, and determine if the concept-in-question is found there. For example, you are interested in the expert's concept "position." You didn't find that concept in the **PATR**icia systems concept report. Then you looked at the **PATR**icia system's keyword report. You didn't find that concept in the keyword report, so you look at the **PATR**icia system's class operational report. In the class operational report, you find a sentence like this "It is possible to paint an object that can be described by an orientation descriptor and a position descriptor and a size descriptor and a style descriptor." Since you find the word "position" in a sentence related to this class, you write "See class operational report" under the expert's concept-in-question on the report-sheet-for-matching, on the line labeled "Same or very similar concepts." **There is NO need to put a checkmark on the class operational report.**

8) Repeat lines 4-7 for all concepts in the current class.

9) For each concept in the **PATR**icia system's concept that report that **does not have a checkmark** after all of the expert's concepts for the class have been handled, write that concept's name on the garbage report sheet, under the current class**. DO NOT WRITE INHERITED CONCEPTS HERE!!!** (They will have been handled earlier).

	A	B	C	D	E	F	G	H	I	J	K	L
1		**List all concepts that were NOT matched to another concept**										0
2												
3	**Watson XWindows**											
4												
5	**Class TAppWindow**											
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												

	A	B	C	D	E	F	G	H	I	J	K	L
26												
27	0											

	A	B	C	D	E	F	G	H	I	J	K	L
1		List all concepts that were not matched to another concept										_
2												
3	GINA											
4												
5	Class GnContracts											
6												
7												
8	Class GnObject											
9												
10												
11	Class GnCommand											
12												
13												
14	Class GnMouseDownCommand											
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												

	A	B	C	D	E	F	G	H	I	J	K	L
26												
27	0											

	A	B	C	D	E	F	G	H	I	J	K	L
1		**List all concepts that were NOT matched to another concept**										0
2												
3	**wxWindows**											
4												
5												
6	**wxObject**											
7												
8												
9	**wxbWindow**											
10												
11												
12	**wxWindow**											
13												
14												
15	**wxbItem**											
16												
17												
18	**wxItem**											
19												
20												
21	**wxbButton**											
22												
23												
24	**wxButton**											
25												

	A	B	C	D	E	F	G	H	I	J	K	L
26												
27	0											

	A	B	C	D	E	F	G	H	I	J	K	L
1	**GINA**											0
2												
3	**Class GnContracts**											
4	**NONE**											
5												
6	**Class GnObject**											
7												
8	**1) object**											
9	Same or very similar concepts											
10	Related concepts											
11												
12	**Class GnCommand**											
13												
14	**1) cursor**											
15	Same or very similar concepts											
16	Related concepts											
17	**2) frame**											
18	Same or very similar concepts											
19	Related concepts											
20												
21	**3) object**											
22	Same or very similar concepts											
23	Related concepts											
24												
25	**4) text**											

	A	B	C	D	E	F	G	H	I	J	K	L
26	Same or very similar concepts											
27	Related concepts											
28												
29	**5) timer_event_messages**											
30	Same or very similar concepts											
31	Related concepts											
32												
33	**6) view**											
34	Same or very similar concepts											
35	Related concepts											
36												
37	**7) xposition**											
38	Same or very similar concepts											
39	Related concepts											
40												
41	**8) yposition**											
42	Same or very similar concepts											
43	Related concepts											
44												
45												
46	**Class GnMouseDownCommand**											
47												
48	**1) button**											
49	Same or very similar concepts											
50	Related concepts											

	A	B	C	D	E	F	G	H	I	J	K	L
51												
52	**2) button_release**											
53	Same or very similar concepts											
54	Related concepts											
55												
56	**3) constrain_mouse**											
57	Same or very similar concepts											
58	Related concepts											
59												
60	**4) cursor**											
61	Same or very similar concepts											
62	Related concepts											
63												
64	**5) down**											
65	Same or very similar concepts											
66	Related concepts											
67												
68	**6) draw_feedback**											
69	Same or very similar concepts											
70	Related concepts											
71												
72	**7) motion_notify**											
73	Same or very similar concepts											
74	Related concepts											
75												

	A	B	C	D	E	F	G	H	I	J	K	L
76	**8) move_mouse**											
77	Same or very similar concepts											
78	Related concepts											
79												
80	**9) move_mouse_down**											
81	Same or very similar concepts											
82	Related concepts											
83												
84	**10) mouse_down**											
85	Same or very similar concepts											
86	Related concepts											
87												
88	**11) mouse_event_messages**											
89	Same or very similar concepts											
90	Related concepts											
91												
92	**12) mouse_idle**											
93	Same or very similar concepts											
94	Related concepts											
95												
96	**13) scrolling_text**											
97	Same or very similar concepts											
98	Related concepts											
99												
100	**14) text**											

	A	B	C	D	E	F	G	H	I	J	K	L
101	Same or very similar concepts											
102	Related concepts											
103												
104	**15) text_scrolling**											
105	Same or very similar concepts											
106	Related concepts											
107												
108	**16) timer_event_messages**											
109	Same or very similar concepts											
110	Related concepts											
111												
112	**17) track_mouse**											
113	Same or very similar concepts											
114	Related concepts											
115												
116	**18) view**											
117	Same or very similar concepts											
118	Related concepts											
119												
120	**19) window**											
121	Same or very similar concepts											
122	Related concepts											
123												
124	**20) window_frame**											
125	Same or very similar concepts											

	A	B	C	D	E	F	G	H	I	J	K	L
126	Related concepts											
127												
128	**21) xposition (xcoordinate)**											
129	Same or very similar concepts											
130	Related concepts											
131												
132	**22) yposition (ycoordinate)**											
133	Same or very similar concepts											
134	Related concepts											

APPENDIX G

REUSABILITY ANALYSIS EXPERIMENT -- DIRECTIONS AND QUESTIONNAIRES

Directions

This experiment attempts to determine the reusability of portions of three separate packages of C++ GUI software. You, along with others, will serve as a C++ expert, and will answer several questionnaires relating to various aspects of the reusability of the given software. Then you will rate the software as reusable on a scale where Excellent = 100%, Good = 75%, Fair = 50%, and Poor = 25%. The purpose of the questionnaire is to ensure that all experts consider the same factors when deciding whether a class or class hierarchy is reusable or not; however, you are allowed to add additional factors that do not appear on the questionnaire. Similarly, if you disagree with part of the questionnaire you are allowed to say so, and state your reasons why.

You will be rating 18 classes over three separate C++ GUI packages. You will also be rating 6 different hierarchies (some of the classes appear in multiple hierarchies). You will look at the class first as a standalone entity, and then as part of a hierarchy. The quality factor framework related to the reusability of a single class is called **Reusability-in-the-class**. The quality factor framework related to the reusability of a set of classes in a single class hierarchy is called **Reusability-in-the-Hierarchy**. Your directions also include a brief description of the quality factor frameworks Reusability-in-the-Original-System, and Reusability-in-the-Subsystem, for completeness; we will not be concerned with the latter two quality factor frameworks in this experiment.

The information on reusability derived from this experiment will be used to test the metrics and quality metrics portion of the **PATR**icia system (**P**rogram **A**nalysis **T**ool for **R**euse), which forms part of my doctoral dissertation. However, you are not concerned directly with the **PATR**icia system in this experiment.

For those of you who have served as experts for me before, the same classes are to be examined in this experiment as were examined in the earlier C++ experiment. However, this time I have included the member function code along with the class definitions for all the classes. This notebook has been reorganized somewhat, with the wxWindows portion placed last. As you recall, the wxWindows portion was the longest. I wish to remind you that the first hierarchy in the wxWindows section is the most important to me of the four hierarchies; however, after the first hierarchy is completed only 6 additional classes remain to be examined (several classes are repeated between the various hierarchies, and each class need only be examined once).

You are provided a set of **Reusability-in-the-Class** questionnaires, at least one each for the 18 classes, plus a couple of extras in case of errors. You are provided one **Reusability-in-the-Hierarchy** questionnaire for each hierarchy. The **Reusability-in-the-Hierarchy** questionnaires are specific to the particular hierarchy, whereas the **Reusability-in-the-Class** questionnaires are interchangeable.

In regard to certain questions on the questionnaires: when I ask, for example, "what is the average number of executable semicolons," or "what is the average number of commented methods," I am not necessarily interested in your laboriously counting each comment or semicolon (although you may if you wish, this would add excessively to the time you spent on this task, and that is not my goal). What I want you to do is to consider these questions when trying to decide, based on your own C++ and GUI experience, whether or not the class (or hierarchy) could be reused with an appropriate amount of effort in a particular project that requires a GUI interface. The listed questions are provided for two reasons: 1) to make certain that you don't forget anything important in your determination of reusability, and 2) to make sure that all my experts

are at least thinking generally in the same way, so that results combining your efforts will be somewhat meaningful.

I estimate the time for this project should take somewhere between 3-5 hours of actual work time. This could vary depending on how carefully you count comments and semicolons, etc., and on your level of C++ experience. Remember that I want your best estimate of reusability, in no more than the amount of time you would spend on a particular class or hierarchy if you were initially considering its use in a project, that is, an amount of time in which you would determine: a) the class (or hierarchy) is worth being looked into further, or b) the class (or hierarchy) is not worthy of being looked at further. You might work from the assumption that you were given these three different packages as possibilities for using in your project, and that you are to decide between them. One caveat I might mention is that all of these packages have been reused somewhat. They are not commercial packages; rather, they are produced by research organizations (located in different parts of the world!). However, the packages have seen varying amounts of use.

The organization of the notebook is as follows: there are three sections, one for each C++ GUI package. GINA is first, followed by Watson Xwindows, then finally comes wxWindows. Each section begins with an index divider that has a green tab with the name of the current C++ GUI package. Within each section: first after the package index divider comes the class hierarchy listing. Then immediately follow the .h files that contain the class definitions. Each .h file is separated from another .h file by a piece of colored construction paper--usually yellow, but in some cases black. Following the .h files comes an index divider that says either GINA .C files, Watson .cxx file, or wxWindow .cc files, as appropriate for the current section. Within these .cc files are the member function definitions for all non-inline functions. Each of the code files (.C, .cxx, or .cc as appropriate) is separated from the other code files by an index divider. The index dividers contain the names of the code files. Beside each class definition in a .h file, hand-written, is the name of the corresponding code file (.C, .cxx, or .cc as appropriate) that contains that class

definitions' member functions.In the case of wxWindows, some class definitions have code that is found in multiple .cxx files. In this case the name of each file is written beside the class definition. Additionally, within the .cxx file, a colored tab with the name of the classes whose member functions are being defined is stuck on the page at which those member function definitions begin. Note that this was necessary only for the wx_item.cc and wb_item.cc files. In the case of the wxWindows package alone, between the class hierarchy at the beginning of the section, and the following .h files, is a page that contains a mapping from .h files to their associated .cxx files.

Note that all class definitions and member functions of interest are highlighted.

As you recall, I plan to publish at least two more conference papers, at conferences that publish proceedings. The name of experts who work on this experiment will appear as co-authors on these conference papers. Those of you who participated in the original experiment already had your names appear on an IEEE conference paper. I will keep you informed as to which conferences I have sent a paper to, and I will send you copies of each paper.

I will be glad to pay any and all postal fees. Just let me know how much.

If you need to get in touch with me with a question, feel free any time. This experiment is very important to me. My addresses, etc., are included for your convenience. Thank you so much for your time and effort!

Given the following set of reuse views as a guideline, answer the given questionnaires.

Reusability-in-the-Class a view of the class taken alone

- **Modularity**
  - **Cohesion**
  - **Coupling**
- **Interface Complexity**
- **Documentation**
- **Simplicity**
  - **Size**
  - **Complexity**

Reusability-in-the-Hierarchy -- a view of a class and its associated direct ancestors

- **Average Modularity**
  - **Average Cohesion**
  - **Average Coupling**
- **Average Interface Complexity**
- **Average Documentation**
- **Average Simplicity**
  - **Average size**
  - **Average complexity**
- **Inheritance Complexity**

Reusability-in-the-Original-System -- a view of the usage of a class in the original system, as a predictor of use in another system

- **Abstract Data Type Usage**
- **Inheritance Usage**

Reusability-in-the-Subsystem -- a view of a selected subsystem of classes

Average Modularity

Average Cohesion

Average Coupling

Average Interface

Average Documentation

Average Simplicity

Average size

Average complexity

Maximum Inheritance Complexity

Reusability-in-the-Class - a view of the class taken alone

Modularity -- Those characteristics of the class which provide for high cohesion and optimum coupling.

Cohesion -- A description of the logical relationship between elements of a class.

Coupling -- A description of the relationship of this class with other software entities (classes, functions, etc.)

Interface Complexity -- The complexity of the interface to a class--the ease of understanding the interface.

Documentation -- Those attributes of a class which provide explanation of the functionality and implementation of the class.

Simplicity -- Those attributes of the class which provide for the definition and implementation of the class in the most non-complex and understandable manner

Size -- The size of the class relates to simplicity in that the larger the class the less simple the class.

Complexity -- The complexity of the class relates to simplicity in that the more complex the class, the less simple the class.

Note: Normally a call to an *inherited* member function would not be counted against a class (would be "good" coupling as opposed to "bad" coupling). Calls to the member functions of classes *not* in the class' direct inheritance hierarchy (not an ancestor class) would be considered part of coupling ("bad" coupling). Calls to *standalone* functions would be considered part of coupling ("bad" coupling). The use of other classes as *abstract data types* within the current class would be considered part of coupling ("bad" coupling).

Note: The above criterion definitions were adapted from Boeing/RADC criterion definitions into definitions that would apply to object-oriented code. If you need a further description of the above criteria, you can refer to the Appendix.

Appendix

From Bowen, T.P., Wigle, G.B., and Tsai, J.T., Specification of Software Quality Attributes, Software Quality Specification Guidebook, Boeing Aerospace Company, RADC-TR-85-37, Vol. II (of three):

Modularity -- Those characteristics of software which provide a structure of highly cohesive components with optimum coupling. (pp. 3-12)

Simplicity -- Those characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner. (pp. 3-12)

Self-descriptiveness -- Those characteristics of software which provide explanation of the implementation of functions. (pp. 3-12)

From Presson, P.E., Tsai, J., Bowen, T.P, Post, J.V., and Schmidt, R., Software Interoperability and Reusability, Boeing Aerospace Company, RADC-TR-83-1784, Vol. I (of two):

Modular Implementation -- Those attributes of the software that provide a structure of highly independent modules. (p. 23)

Software that is constructed in a modular fashion will tend to limit the impact of changes necessary for interoperability, thus making the modifications easier to assess and implement. (p. 44)

Module coupling -- the level of interdependence between modules within a system. (p. 81)

Modularity -- Software possesses the characteristics of modularity to the extent that a logical partitioning of software into independent parts, components, and modules has occurred. (p. 93)

Modularity is the concept of confining specific design decisions or functions into a distinct design element which is as independent of other elements as possible. This independence helps to localize the impact of modifications to within one or a few modules. The advantage of such a modularly designed system is that modules can be replaced or modified without disturbing system functions so long as their interface meets the stated requirements. The primary design goal is to produce a design of the modular structure of the program so that the modules are highly independent. The parts of a module join forces to perform a single specific well-defined function and the data should be explicitly passed as parameters or arguments. In other words we want to maximize the relationships among parts of each module (module strength) and to minimize the relationships among modules (module coupling). (p. 97)

The Modularity of software is the key factor of module, function, and partial reuse. Since this type of reuse entails interfacing of the existing software with new software, the modularity of the existing software will determine the cost of reusing the software. (p. 101)

Simplicity -- If the design structure of the software system is simple, it will be easier for the software engineer to understand and thus easier to modify. (p. 45)

Simplicity -- Software possesses the characteristics of simplicity to the extent that it lacks complexity in organization, language, and implementation techniques and is constructured in the most understandable manner. (p. 94)

Simplicity -- The concept of simplicity implies that the software is lacking in complexity. The more basic techniques, structures, etc., are used, the simpler the software will tend to be. The quantitative counts (number of operators, operands, nested control structures, nested data structures, executable statements, statement labels, decision points, parameters, etc.) will

determine to a great extent how simple or complex the source code is. Simplicity, in both design and implementation, will tend to make programs simpler, easily understandable, and easily modifiable. (p. 97)

The Simplicity and System Clarity are the important characteristics of software comprehensibility. In the module, function or partial reuse of software, how simple and clear the software is will determine the costs to reuse the software. (p. 101)

Self-Descriptiveness -- The quantity and quality of comments in the source code and the level of the language will directly affect the ability of the software engineer to comprehend the software design well enough to implement the modifications necessary for interoperability.

Self-descriptiveness -- Software possesses the characteristics of self-descriptiveness to the extent that it contains information regarding its objectives, assumptions, constraints, inputs, processing, outputs, components, etc. (p. 94)

Self-descriptiveness -- The concept of self-descriptiveness implies that the software contains enough information for the reader to determine or verify its objectives, assumptions, constraints, inputs, processing, outputs, components, etc. This quality is very important in being able to understand the software. (p. 98)

The Self-Descriptiveness is very important in being able to understand the software. For the module, function, or partial reuse of the software, there is a necessity of completely understanding the software. (p. 101)

Interface complexity -- The ease of understanding module interfaces, in terms of the data affected. (p. 81)

Reusability of software requires that the software be understandable, flexible, modifiable, adaptable, applicable, and accessible. Simplicity, system clarity, and self-descriptiveness criteria

will enhance the understandability. Generality, machine and software system independences, application independence and modularity will improve the flexibility, modifiability, and adaptability. Well-structured documentation and no-control access will improve the accessibility. (p. 5)

From Bowen, T.P., Wigle, G.B., and Tsai, J.T. Specification of Software Quality Attributes: Software Quality Evaluation Guidebook, RADC-TR-85-37, Vol. III (of three), Feb. 1985:

Modularity -- Those characteristics of software which provide a structure of highly cohesive components with optimum coupling. (pp. 3-14)

Self-descriptiveness -- Those characteristics of software which provide explanation of the implementation of functions. (pp. 3-14)

Simplicity -- Those characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner. (pp. 3-14)

Cohesion Value: The type of relationship that exists among the elements of each software entity. There are seven types of cohesion (pp. A-180-A-181):

1) coincidental: No meaningful relationships among the elements of an entity. Difficult to describe the module's function(s).

2) logical: Entity (at each invocation) one of a class of related functions (e.g., "edit all data"). Entity performs more than one function.

3) classical: Entity performs one of a class of functions that are related in time (program procedure). Entity performs more than one function.

4) procedural: Entity performs more than one function, where the functions are related with respect to the procedure of the problem (Problem procedure).

5) Communicational: Entity has procedural strength; in addition, all of the elements "communicate" with one another (e.g. reference same data or pass data among themselves). All functions use the same data.

6) Informational: Entity performs multiple functions where the functions (entry points in the unit) deal with a single data structure. Physical packaging together of two or more entities having functional strength. All functions use the same data.
7) Functional: All entity elements are related to the performance of a single function.

Coupling: The type of relationship that exists between two software entities. In achieving a highly modular design it is essential to minimize the relationships among software entities. The goal is to design software entities with low coupling. The scale of coupling from worst to best is: 1) Content coupling, 2) Common Coupling, 3) External Coupling, 4) Control Coupling, 5) Stamp Coupling, and 6) Data Coupling.
1) Content Coupling -- One software entity references the contents of another software entity.
2) Common Coupling -- Software entities reference a shared global data structure.
3) External Coupling -- Software entities reference the same externally declared symbol.
4) Control Coupling -- One software entity passes control elements as arguments to another software entity.
5) Stamp Coupling -- Two software entities reference the same data structure, which is not global.
6) Data Coupling -- One software entities calls another and the software entities are not coupled as defined above (in 1 through 5). (pp. A-181-A-182)

Reusability-in-the-hierarchy

Uses the same per-class definitions as those shown in the Reusability-in-the-Class view. However, the values for each criterion should represent an average over all classes in the hierarchy.

Average Modularity -- Those attributes of the class which provide for high cohesion and optimum coupling, for all classes in the hierarchy.

Average Cohesion -- A description of the logical relationship between elements of a class, for all classes in the hierarchy.

Average Coupling -- A description of the relationship of this class with other software entities (classes, functions, etc.) for all classes in the hierarchy.

Average Interface Complexity -- The complexity of the interface to a class for all classes in the hierarchy.

Average Documentation -- Those attributes of a class which provide explanation of the functionality and implementation of the class, for all classes in the hierarchy.

Average Simplicity --Those attributes of the class which provide for the definition and implementation of the class in the most non-complex and understandable manner, for all classes in the hierarchy.

Average Size -- The size of the class relates to simplicity in that the larger the class the less simple the class. Determined for all classes in the hierarchy.

Average Complexity -- The complexity of the class relates to simplicity in that the more complex the class, the less simple the class. Determined for all classes in the hierarchy.

Additionally, it provides:

Inheritance Complexity -- Those attributes of a class which relate to complexity derived from the fact that the class exists in an inheritance hierarchy.

Reusability-in-the-Class Questionnaire

MODULARITY

Cohesiveness:

CH1) How big is the class in terms of number of attributes and number of methods? (a large class is less likely to be cohesive)

number of attributes: _____.

number of methods: _____.

CH2) Do the methods in the class use disjoint sets of attributes (do there exist methods that have no attributes in common with other methods -- this could be a hint that the class should be broken into two or more classes)?

# of disjoint sets of methods: _____.

CH3) Are the methods in the class closely related in functionality?

Very related, somewhat related, not related

(circle one)

CH4) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **cohesiveness**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of cohesiveness:

Please weight the criteria according to their importance to the determination of cohesiveness. The weights should add up to 100%. For example, you might weight them equally, with CH1 being 33.3%, CH2 33.3%, and CH3 33.3%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Remember to weight any additional criteria.

Weights:

CH1: _____.

CH2: _____.

CH3: _____.

other#1: _____.

other#2: _____.

Coupling:

CO1) Do any class methods use global data?

number of times global data is used: _____.

CO2) Does the class have any friend functions or classes?

number of friends: _____.

CO3) Do the class methods access the attributes of any other class not in the class' direct hierarchy (list of direct ancestor classes)?

number of times attributes of other classes accessed: _____.

CO4) Do the class methods access the methods of any other class not in the class' direct hierarchy (list of direct ancestor classes)?

number of times methods of other classes accessed: _____.

CO5) Do the class methods access any external-to-the-class (standalone) functions (except for methods of other classes)?

number of times external functions accessed: _____.

CO6) Are there any variable definitions, either in the class definition, or local to a member function, that use another class as an abstract data type?

number of abstract data types: _____.

CO7) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **coupling**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of coupling:

Please weight the criteria according to their importance to your determination of coupling. The weights should add up to 100%. For example, you might weight them equally, with CO1 being 16.67%, CO2 16.67%, CO3 16.67%, CO4 16.67%, C05 16.67, and CO6 16.67%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Remember to weight any additional criteria.

Weights:

CO1: _____.

CO2: _____.

CO3: _____.

CO4: _____.

CO5: _____.

CO6: _____.

other #1: _____.

other #2: _____.

M1) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **modularity**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of modularity:

Please weight the criteria according to their importance to your determination of modularity. The weights should add up to 100%. For example, you might weight them equally, with CH4 being 50%, and CO6 50%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Remember to weight any additional criteria.

Weights:

CH4: _____.

CO7: _____.

other#1: _____.

other#2: _____.

INTERFACE

I1) How many public methods are there?

number of public methods: _____.

I2) How many formal parameters, on the average, do the public method definitions have?

average number of formal parameters in public methods: _____.

I3) Are the public methods at the appropriate granularity level? That is, do they do too much, not enough, or too little for the functionality provided by the class? Should some of their required functionality be moved to an internal private method, then that method be called by the public method?

good granularity level, fair granularity level, too much in the method, too little in the method

(circle one)

I5) Are the public methods clean and easy to understand?

Very easy to understand, Fairly easy to understand, Somewhat Easy to Understand, Hard to understand

(circle one)

I6) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **interface**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of the quality of the interface:

Please weight the criteria according to their importance to the determination of the quality of the interface. The weights should add up to 100%. For example, you might weight them equally, with I1 being 20%, I2 20%, I3 20%, I4 20%, and I5 20%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Remember to weight any additional criteria.

Weights:

I1: _____.

I2: _____.

I3: _____.

I4: _____.

I5: _____.

other#1: _____.

other#2: _____.

DOCUMENTATION

D1) How many comments are there in the class definition?

number of comments: _____.

D2) How many comments are there in each method, on the average?

average number of comments: _____.

D3) What percentage of methods have any comments at all?

percentage of commented methods: _____.

D4) Are the comments in general well-written, understandable, and meaningful?

Excellent, Good, Fair, Poor

(circle one)

D5) Are the identifier names (class names, variable names, method names, etc.) well-chosen, understandable, and meaningful?

Excellent, Good, Fair, Poor

(circle one)

D6) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **documentation**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of the quality of the documentation:

Please weight the criteria according to their importance to the determination of documentation quality. The weights should add up to 100%. For example, you might weight them equally, with D1 being 20%, D2 20%, D3 20%, D4 20%, and D5 20%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Remember to weight any additional criteria.

Weights:

D1: _____.

D2: _____.

D3: _____.

D4: _____.

D5: _____.

other#1: _____.

other#2: _____.

SIMPLICITY

Size:

SI1) How big is the class in terms of number of attributes and number of methods?

number of attributes (variables): _____.

number of methods (member functions): _____.

SI2) How many lines of executable semicolons are there in the class definition (ignoring comments, blank lines, etc.)

number of executable semicolons: _____.

SI3) How many executable semicolons are there, on the average, in the methods?

average number of executable semicolons: _____.

SI4) How many formal parameters are there in the method definitions, on the average?

average number of formal parameters: _____.

SI5) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **size**.

Large, medium, small

(circle one)

Please list and describe all other criteria that you thought was important in your determination of size:

Please weight the criteria according to their importance to the determination of size. The weights should add up to 100%. For example, you might weight them equally, with S1 being 25%, S2 25%, S3 25%, and S4 25%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Please remember to weight any additional criteria.

Weights:

SI1: _____.

SI2: _____.

SI3: _____.

SI4: _____.

other#1: _____.

other#2: _____.

Complexity:

C1) How complex is the code in the methods, on the average?

Very complex, Fairly complex, Not complex

(circle one)

Please list and describe all other criteria that you thought was important in your determination of complexity:

Please weight the criteria according to their importance to the determination of complexity. The weights should add up to 100%. For example, you might weight C1 at 100%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Please remember to weight any additional criteria.

Weights:

C1: _____.

other#1: _____.

other#2: _____.

S1) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **simplicity**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of simplicity:

Please weight the criteria according to their importance to the determination of the simplicity. The weights should add up to 100%. For example, you might weight them equally, with SI5 being 50% and C1 being 50%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Please remember to weight any additional criteria.

Weights:

SI5: _____.

C1: _____.

other#1: _____.

other#2: _____.

Reusability-in-the-Class

R1) Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **Reusability-in-the-Class**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of simplicity:

Please weight the criteria according to their importance to the determination of the reusability of the class. The weights should add up to 100%. For example, you might weight them equally, with Modularity being 25%, Interface 20%, Documentation 20%, and Simplicity 20%, if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Please remember to weight any additional criteria.

Weights:

Modularity(M1): _____.

Interface(I6): _____.

Documentation(D6): _____.

Simplicity(S1): _____.

other#1: _____.

other#2: _____.

Reusability-in-the-Hierarchy

RH1) Determine average modularity by first looking at average cohesion and average coupling, then by looking at the modularity values previously determined for each class.

RH1a) Look at the cohesion values for each class in the hierarchy.

CH4: class #1: _____. class #11: _____.

class #2: _____. class #12: _____.

class #3: _____. class #13: _____.

class #4: _____. class #14: _____.

class #5: _____. class #15: _____.

class #6: _____. class #16: _____.

class #7: _____. class #17: _____.

class #8: _____. class #18: _____.

class #9: _____. class #19: _____.

class #10: _____. class #20: _____.

Now, rate the hierarchy for **average cohesion**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

RH1b) Look at the coupling values for each class in the hierarchy.

CO7: class #1: _____. class #11: _____.

class #2: _____. class #12: _____.

class #3: _____. class #13: _____.

class #4: _____. class #14: _____.

class #5: _____. class #15: _____.

class #6: _____. class #16: _____.

class #7: _____. class #17: _____.

class #8: _____. class #18: _____.

class #9: _____. class #19: _____.

class #10: _____. class #20: _____.

Now, rate the hierarchy for **average coupling**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Finally, look at the modularity values for each class in the hierarchy:

MO1: class #1: _____. class #11: _____.

class #2: _____. class #12: _____.

class #3: _____. class #13: _____.

class #4: _____. class #14: _____.

class #5: _____. class #15: _____.

class #6: _____. class #16: _____.

class #7: _____. class #17: _____.

class #8: _____. class #18: _____.

class #9: _____. class #19: _____.

class #10: _____. class #20: _____.

Now, rate the hierarchy for **average modularity**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

RH2) Look at the interface complexity values for each class in the hierarchy:

I6:	class #1:	_____.	class #11:	_____.
	class #2:	_____.	class #12:	_____.
	class #3:	_____.	class #13:	_____.
	class #4:	_____.	class #14:	_____.
	class #5:	_____.	class #15:	_____.
	class #6:	_____.	class #16:	_____.
	class #7:	_____.	class #17:	_____.
	class #8:	_____.	class #18:	_____.
	class #9:	_____.	class #19:	_____.
	class #10:	_____.	class #20:	_____.

Now, rate the hierarchy for **average interface complexity**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

RH3) Look at the documentation values for each class in the hierarchy:

D6:	class #1:	_____.	class #11:	_____.
	class #2:	_____.	class #12:	_____.
	class #3:	_____.	class #13:	_____.
	class #4:	_____.	class #14:	_____.
	class #5:	_____.	class #15:	_____.
	class #6:	_____.	class #16:	_____.
	class #7:	_____.	class #17:	_____.
	class #8:	_____.	class #18:	_____.
	class #9:	_____.	class #19:	_____.
	class #10:	_____.	class #20:	_____.

Now, rate the hierarchy for **average documentation**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

RH4) Determine average simplicity by first looking at average size and average complexity, then by looking at the simplicity values previously determined for each class.

Look at the size values for each class in the hierarchy:

SI5:

class #1:	_____.	class #11:	_____.
class #2:	_____.	class #12:	_____.
class #3:	_____.	class #13:	_____.
class #4:	_____.	class #14:	_____.
class #5:	_____.	class #15:	_____.
class #6:	_____.	class #16:	_____.
class #7:	_____.	class #17:	_____.
class #8:	_____.	class #18:	_____.
class #9:	_____.	class #19:	_____.
class #10:	_____.	class #20:	_____.

Now, rate the hierarchy for **average size**.

Large, medium, small

(circle one)

Look at the complexity values for each class in the hierarchy:

C1:

class #1:	_____.	class #11:	_____.
class #2:	_____.	class #12:	_____.
class #3:	_____.	class #13:	_____.
class #4:	_____.	class #14:	_____.
class #5:	_____.	class #15:	_____.
class #6:	_____.	class #16:	_____.
class #7:	_____.	class #17:	_____.
class #8:	_____.	class #18:	_____.
class #9:	_____.	class #19:	_____.
class #10:	_____.	class #20:	_____.

Now, rate the hierarchy for **average complexity**.

Very complex, Fairly complex, Not complex

(circle one)

Finally, look at the simplicity values for each class in the hierarchy:

S1:

class #1:	_____.	class #11:	_____.
class #2:	_____.	class #12:	_____.
class #3:	_____.	class #13:	_____.
class #4:	_____.	class #14:	_____.
class #5:	_____.	class #15:	_____.
class #6:	_____.	class #16:	_____.
class #7:	_____.	class #17:	_____.
class #8:	_____.	class #18:	_____.
class #9:	_____.	class #19:	_____.
class #10:	_____.	class #20:	_____.

Now, rate the hierarchy for **average simplicity**.

Excellent (= 100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

RH5) Look at the depth of the inheritance tree of the class. An excessive depth of inheritance would tend to mean that the lower classes in the inheritance hierarchy would be quite complex (since they would inherit all the complexity of their direct ancestor classes).

depth of inheritance tree: _____.

RH6) Now, using the criteria in the above questions, and any other criteria you think is appropriate, rate the hierarchy for **Reusability-in-the-hierarchy.**

Excellent (=100%), Good (= 75%), Fair (= 50%), Poor (= 25%)

(circle one)

Please list and describe all other criteria that you thought was important in your determination of simplicity:

Please weight the criteria according to their importance to the determination of the reusability of the hierarchy. The weights should add up to 100%. For example, you might weight them equally, with Average Modularity being 20%, Average Interface 20%, Average Documentation 20%, Average Simplicity 20%, and Depth of Inheritance Tree 20% if you used no additional criteria. If you don't think a criterion is useful, weight it at 0%. Please remember to weight any additional criteria.

Weights:

Average Modularity(RH1)): _____.

Average Interface(RH2): _____.

Average Documentation(RH3): _____.

Average Simplicity(RH4): _____.

Depth of Inheritance Tree(RH5): _____.

other#1: _____.

other#2: _____.

REFERENCES

[1] Abd-El-Hafiz, S.K., Basili, V.R., "A Knowledge-Based Approach to the Analysis of Loops," *IEEE Transactions on Software Engineering*, Vol. 22, No. 5, May, 1996, pp. 339-360.

[2] Abi-Akar, R. *An Investigation of Programming Language Mechanisms for Software Reuse*, doctoral dissertation, The University of Alabama in Huntsville, 1992.

[3] Allen, J., *Natural Language Understanding*, The Benjamin/Cummings Publishing Company, Redwood City, CA, 1995.

[4] Andersen, P.M., Hayes, P.J., Heuttner, A.K., Schmandt, L.M., and Nirenberg, I.B., "Automatic Extraction," *Proceedings of the Conference of the Association for Artificial Intelligence*, Philadelphia, PA, 1986, pp. 1089-1093.

[5] ANSI X3H4. (Draft Proposed) *American National Standard Information Resource Dictionary System: Part 1--Core Standard*, American National Standards Institute, New York, 1985.

[6] Aristar, A., Dry, H., and Seely, D., Eds., *The Linguist List*, www.emich.edu/~linguist, Discussions Number 8.249, 8.135, and 8.66, January, 1997.

[7] 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.

[8] Babatz, R., Baecker, A., *GINA Manual*, Version 2.0, German National Research Institute for Computer Science, anonymous ftp at ftp.gmd.de, directory gmd/ginaplus, 1991.

[9] Bansiya, J., and Davis, C.G., "An Object-Oriented Design Quality Assessment Model," submitted to Metrics '97, to be held in Albuquerque, NM., Nov. 5-7, 1997.

[10] Barnden, J.A., "Semantic Networks," *The Handbook of Brain Theory and Neural Networks*, The MIT Press, 1995.

[11] Basili, V.R., Briand, L, and Melo, W.L., "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, October, 1996, pp. 751-761.

[12] Belady, L.A., and Lehman, M.M., "A Model of Large Program Development," *IBM Systems Journal*, Vol. 15, No. 3, 1976, pp. 225-252.

[13] Bickerton, D. and Bralich, P., *Ergo Linguistic Technologies web page*, www.ergo-ling.com.

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

[15] Biggerstaff, T.J., "Human-Oriented Conceptual Abstractions in the Re-engineering of Software," *Proceedings of the Twelfth International Conference on Software Engineering*, March 26-30, 1990, Nice, France, IEEE Computer Society Press, Los Alamitos, CA, 1990, p. 120.

[16] Biggerstaff, T.J., "Design Recovery for Maintenance and Reuse," *IEEE Computer*, Volume 22, Issue 7, July 1989, pp. 36-49.

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

[18] Biggerstaff, T.J., and Richter, C., "Reusability Framework, Assessment, and Directions," *IEEE Software*, Vol. 4, No. 2, March, 1987, pp. 41-49.

[19] Biggerstaff, T.J, Hoskins, J., and Webster, D., "DESIRE: A System for Design Recovery," *MCC Technical Report*, STP-081-89, Microelectronics and Computer Technology Corporation, April, 1989.

[20] Biggerstaff, T.J., Mitbander, B.G, and Webster, D.E, "Program Understanding and the Concept Assignment Problem," *Communications of the ACM*, Volume 37, Number 5, May 1994, pp. 72-82.

[21] Biggerstaff, T.J., Mitbander, B.G., and Webster, D.E., "The Concept Assignment Problem in Program Understanding," *Proceedings of the Fifteenth International Conference on Software Engineering*, May 17-21, 1993, Baltimore, MD, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 482-498.

[22] Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, The Benjamin/Cummings Publishing Company, Inc., 1994.

[23] Brooks, R., "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man and Machine Studies*, Vol. 18, 1983, pp. 543-554.

[24] Caldiera, G., and Basili, V.R., "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, Feb. 1991, pp. 61-70.

[25] Cavaliere, M.H., "Reusable Code at the Hartford Insurance Group," *Software Reusability, Vol. II, Applications and Experience*, ed. T.J. Biggerstaff and A.J. Perlis. ACM Press, Addison-Wesley, Reading, Mass., 1989, pp. 47-76.

[26] 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.

[27] Chidamber, S.R., and Kemerer, C.F., "Towards a Metrics Suite for Object-Oriented Design," *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '91*), 1991, pp. 97-211.

[28] *CLIPS User's Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, May 1993.

[29] *CLIPS Reference Manual, Volume I, Basic Programming Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, June 1993.

[30] *CLIPS Reference Manual, Volume II, Advanced Programming Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, June 1993.

[31] *CLIPS Reference Manual, Volume III, Interfaces Guide*, Software Technology Branch, Lyndon B. Johnson Space Center, June 1993.

[32] Collefello, J.S., and Bortman, S., "An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance," *Proceedings of the Phoenix Conference on Computers and Communications*, Scottsdale, AZ, March 1986, pp. 420-424.

[33] Cowie, J., and Lehnert, W., "Information Extraction," *Communications of the ACM*, Volume 39, Number 1, January 1996, pp. 80-91.

[34] DARPA, *Proceedings of the Third Message Understanding Conference (MUC-3)*, San Diego, CA, May 21-23, 1991, Morgan Kaufmann, 1991.

[35] DARPA, *Proceedings of the Fourth Message Understanding Conference (MUC-4)*, McLean, VA, June, 1992, Morgan Kaufmann, San Mateo, CA, 1992.

[36] DeJong, G., "An Overview of the FRUMP System," *Strategies for Natural Language Processing*, edited by M. Ringle and W. Lehnert, Lawrence Erlbaum Associates, 1982, pp. 149-176.

[37] Demarco, T., *Structured Analysis and System Specifications*, Prentice-Hall, 1979.

[38] Duncan, R.P., *A Presentation Environment for Improving Software Understanding*, Master's Thesis, The University of Alabama in Huntsville, 1991.

[39] Dunn, M.F., and Knight, J.C., "Automating the Detection of Reusable Parts in Existing Software," *Proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD, May 17-21, 1993, IEEE Computer Society Press, Los Alamitos, CA, pp. 381-390.

[40] Etzkorn, L.H., "A Metrics-based Approach to the Automated Identification of Object-oriented Reusable Components: A Short Overview," *OOPSLA '95 Doctoral Symposium*, Austin, TX, October 16-19, 1995.

[41] Etzkorn, L.H., and Davis, C.G. "Automatically Identifying Reusable Components in Object-Oriented Legacy Code," *IEEE Computer*, (accepted, to appear October, 1997).

[42] Etzkorn, L.H., and Davis, C.G., "Automated Object-oriented Reusable Component Identification," *Knowledge-Based Systems*, Volume 9, Issue 8, Elsevier Publishing, Oxford, England, Dec. 1996, pp. 517-524.

[43] Etzkorn, L.H., and Davis, C.G., "A Metrics-based Approach to the Automated Identification of Object-oriented Reusable Components: A Short Overview," *OOPSLA '95 poster session*, Austin, TX, October 16, 1995.

[44] Etzkorn, L.H., and Davis, C.G., "Automated Object-oriented Reusable Component Identification," *Second International Symposium on Knowledge Acquisition, Representation and Processing*, Auburn, AL, September 27-30, 1995, later republished in Knowledge-Based Systems, Elsevier Publishing.

[45] 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.

[46] 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.

[47] Etzkorn, L.H., and Davis, C.G., "An Approach to Object-Oriented Program Understanding," Technical Report TR-UAH-CS-1995-01, Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL, Sept. 1995.

[48] Etzkorn, Letha, Davis, Carl, and Li, Wei, "A Practical Look at the Lack of Cohesion in Methods Metric," *Journal of Object-oriented Programming*, (accepted, to appear 1998).

[49] Etzkorn, Letha, Davis, Carl, and Li, Wei, "A Statistical Comparison of Various Definitions of the LCOM Metric," Technical Report TR-UAH-CS-1997-02, Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL, April 1997.

[50] Etzkorn, L.H., Davis, C.G., Bowen, L.L., Etzkorn, D.B., Lewis, L.W., Vinz, B.L., Wolf, J.C., "A Knowledge-Based Approach to Object-Oriented Legacy Code Reuse," *Proceedings of the Second International Conference on Engineering of Complex Computer Systems*, Montreal, Quebec, Canada, Oct. 21-25, 1996, IEEE Computer Society, Los Alamitos, CA, 1996, pp. 39-45.

[51] Etzkorn, L.H., Davis, C.G., Vinz, B.L., Wolf, R.P., Wolf, J.C., Yun, M.Y., Bowen, L.L., Orme, A.M., Lewis, L.W., and Etzkorn, D.B., "An Examination of Object-Oriented Reuse Views in the **PATR**icia System," Technical Report TR-UAH-CS-1997-01, Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL, January 1997.

[52] Fitzpatrick, E., Bachenko, J., and Hindle, D., "The Status of Telegraphic Sublanguages," *Analyzing Language in Restricted Domains*, Grishman, R., and Kittredge, R. Eds., Lawrence Erlbaum Associates, New Jersey, 1986, pp. 39-51.

[53] Frakes, W., and Nejmeh, B., "Software Reuse Through Information Retrieval," *Proceedings of the Twentieth Annual Hawaii International Conference on System Science*,eds. B.D.Shriver and R.H.Sprague, Jr., Kailua-Kona, Hawaii, IEEE Computer Society, Los Alamitos, CA, 1987, pp. 530-535.

[54] GPALS Computer Resource Working Group (CRWG) Software Reuse Committee, "GPALS Software Reuse Strategy," February 1992.

[55] Grishman, R., and Kittredge, R., Eds., *Analyzing Language in Restricted Domains*, Lawrence Erlbaum and Associates, New Jersey, 1986.

[56] Halstead, M., *Elements of Software Science*, North Holland, 1977.

[57] Harandi, M.T., and Ning, J.Q., "PAT: A Knowledge-Based Program Analysis Tool," *Proceedings of the Conference on Software Maintenance*, Scottsdale, AZ, Oct. 24-27, 1988, IEEE Computer Society Press, Washington, DC, pp. 312-318.

[58] Hitz, M., and Montazeri, B., "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE Transactions on Software Engineering*, Vol. 22, No. 4, April 1996, pp. 267-271.

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

[60] *International Mathematics and Scientific Library, Tenth Edition*, IMSL, Inc., 1984.

[61] Jette, C. and Smith, R. "Examples of Reusability in an Object-Oriented Programming Environment," *Software Reusability, Vol. II, Concepts and Models*, eds. T.J. Biggerstaff and A.J. Perlis, ACM Press, Addison-Wesley, Reading, Mass., 1989, pp. 73-101.

[62] Jones, G., and Prieto-Diaz, R., "Building and Managing Software Libraries," *Proceedings of IEEE COMPSAC*, IEEE Computer Society, Los Alamitos, CA, 1988, pp. 228-236.

[63] Kim, Y., and Stohr, E.A., "Software Reuse: Issues and Research Directions," *Proceedings of the Twenty-fifth Hawaii International Conference on System Science*, Kauai, Hawaii, Jan. 7-10, 1992, Vol. 4, IEEE Computer Society, Los Alamitos, CA, pp. 612-623.

[64] Kittredge, R., "Variation and Homogeneity of Sublanguages," *Sublanguage: Studies of Language in Restricted Semantic Domains*, Eds. Kittredge, R., and Lehrberger, J., Walter de Gruyter, New York, 1982, pp.107-137.

[65] Kittredge, R., and Lehrberger, J., *Sublanguage: Studies of Language in Restricted Semantic Domains*, Walter de Gruyter, New York, 1982.

[66] Korson, T., and McGregor, J., "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, Vol. 33, No. 9, pp. 40-60.

[67] Kozaczynski, W., and Ning, J.Q., "Automated Program Understanding by Concept Recognition," *Automated Software Engineering: The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering*, Vol. 1, Kluwer Academic Publishers, Boston, MA, 1994, pp. 61-78.

[68] Kozaczynski, W., Ning, J., Engberts, A., "Program Concept Recognition and Transformation," *IEEE Transactions on Software Engineering*, Volume 18, Issue 12, 1992, pp. 1065-1075.

[69] Kozaczynski, W., Ning, J., and Engberts, A., "Intelligent Programming Constraint Maintenance in Software Reengineering," *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, Chicago, IL, July 6-10, 1992, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 83-94.

[70] Kozaczynski, W., Ning, J., Sarver, T., "Program Concept Recognition," *Proceedings of the Seventh Knowledge-Based Software Engineering Conference*, McLean, VA, Sept. 20-23, 1992, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 216-225.

[71] Lehrberger, J., "Sublanguage Analysis," *Analyzing Language in Restricted Domains*, Eds. Grishman, R., and Kittredge, R., Lawrence Erlbaum Associates, New Jersey, 1986, pp. 19-38.

[72] Lehrberger, J., "Automatic Translation and the Concept of Sublanguage," *Sublanguage: Studies of Language in Restricted Semantic Domains*, Eds. Kittredge, R., and Lehrberger, J., Walter de Gruyter, New York, 1982, pp. 81-106.

[73] Letovsky, S.I., *Plan Analysis of Programs*, doctoral dissertation, Yale University, December, 1988.

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

[75] Li, W., and Henry, S., "Maintenance Metrics for the Object-Oriented Paradigm," *Proceedings of the First International Software Metrics Symposium*, Baltimore, MD, May 21-22, 1993, IEEE Computer Society Press, Los Alamitos, CA, pp. 52-60.

[76] Li, W., Henry, S., Kafura, D., and Schulman, R., "Measuring Object-Oriented Design," *Journal of Object-Oriented Programming*, Volume 8, No. 4, July/August 1995, pp. 48-55.

[77] Lieberherr, Karl J., and Holland, Ian, "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Volume 6, Number 5, September 1989, pp. 38-48.

[78] Lorenz, M., and Kidd, J., *Object-Oriented Software Metrics*, PTR Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[79] Lytinen, S., and Gershman, A., "ATRANS: Automatic Processing of Money Transfer Messages," *Proceedings of the Fifth National Conference of the American Association for Artificial Intelligence*, IEEE Computer Society Press, Los Alamitos, CA, 1993, pp. 93-99.

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

[81] McCabe, T., "A Software Complexity Measure," *IEEE Transaction on Software Engineering*, Vol. 2, No. 6, December, 1976, pp. 308-320.

[82] Ning, J.Q., *A Knowledge Based Approach to Automatic Program Analysis*, doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana, Ill., 1989.

[83] *OOPSLA '96 Advance Program*, Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96), San Jose, CA, Oct. 6-10, 1996.

[84] *OOPSLA '95 Program*, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95), Austin, TX, Oct 16-19, 1995.

[85] Oskarsson, O., "Reusability of Modules with Strictly Local Data and Devices--A Case Study," *Software Reusability, Vol. II, Concepts and Models*, Eds. T.J. Biggerstaff and A.J. Perlis, ACM Press, Addison-Wesley, Reading. Mass., 1989, pp. 143-155.

[86] Ostertag, E.J., and Hendler, J.A. "Computing Similarity in a Reuse Library System: An AI-based Approach," *Technical Report, UMIACS-TR-91-10, CS-TR-2593*, Institute for Advanced Computer Studies, and the Department of Computer Science, the University of Maryland at College Park, January 1991.

[87] Ostertag, E.J., and Hendler, J.A. "An AI-based Reuse System," *Technical Report, UMIACS-TR-89-16, CS-TR-2197*, Institute for Advanced Computer Studies, and the Department of Computer Science, the University of Maryland at College Park, February, 1989.

[88] *PAT - Parallelization Tool*, Georgia Institute of Technology, available via anonymous ftp to ftp.cc.gatech.edu, in directory /pub/people/pat/p2.c++/pat.tar

[89] *PCMETRIC Manual*, SET Laboratories, Mulino, OR, 1994.

[90] Poulin, J., and Caruso, J., "A Reuse Metrics and Return on Investment Model," *Proceedings of the 2nd International Workshop on Software Reusability*, Lucca, Italy, March 24-26, 1993, pp. 75-81.

[91] Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1992.

[92] Presson, P.E., Tsai, T., Bowen, T.P., Post, J.U., and Schmidt, R., "Software Interoperability and Reusability Guidelines for Software Quality Measurement," *RADC-TR-83-174, Volume 2, Final Technical Report*, Boeing Aerospace Company, Rome Air Development Center, Air Force Systems Command, Griffiths Air Force Base, NY, 1983.

[93] Prieto-Diaz, R., and Freeman, P., "Classifying Software for Reusability," *IEEE Software*, Vol.4, No. 1, Jan. 1987, pp. 6-16.

[94] 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.

[95] Rich, C., and Waters, R.C. "The Programmer's Apprentice: A Research Overview," *IEEE Computer*, Volume 21, Issue 11, November, 1988, pp. 12-25.

[96] Rich, C., and Wills, L.M. "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan. 1990, pp. 82-89.

[97] Rich, E., and Knight, K., *Artificial Intelligence*, 2nd Ed., McGraw-Hill, New York, 1991.

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

Measurement," *Proceedings of the IEEE*, volume 77, no.4, April 1989, pp. 581-595.

[99] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[100] Selby, Richard, "Quantitative Studies of Software Reuse," *Software Reusability, Vol. II, Concepts and Models*, Eds. T.J. Biggerstaff and A.J. Perlis, ACM Press, Addison-Wesley, Reading. Mass., 1989, pp. 213-233.

[101] Sleator, D., and Temperley, D., "Parsing English with a Link Grammar," Carnegie Mellon School of Computer Science, CMU-CS-91-196, October 1991, available at www.cs.cmu.edu/~sleator, click on link grammar.

[102] Smart, J., *wxWindows Users Manual*, Version 1.60, Artificial Intelligence Applications Institute, University of Edinburgh, Scotland, UK, http://www.aiai.ed.ac.uk/~jacs/wx, 1994.

[103] Sowa, J. Ed., *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, Morgan Kauffman Publishers, San Mateo, CA, 1991.

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

[105] Swisek, S.E., and Mikkelson, K.W., *Report on the U.S. Software Industry Trends*, performed by Economists, Inc., for the Business Software Alliance, www.bsa.org/info/economicsreport/page1.html.

[106] Tarumi, H., Agusa, K., and Ohno, Y., "A Programming Environment Supporting Reuse of Object-Oriented Software," *Proceedings of the Tenth International Conference on Software Engineering*, Singapore, April, 1988, IEEE Computer Society Press, Los Alamitos, CA, 1988, pp. 265-273.

[107] 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, IEEE Computer Society press, Los Alamitos, CA, USA, 1992, pp. 359-368.

[108] Velardi, P., Pazienza, M.T., and De' Giovanetti, M., "Conceptual Graphs for the Analysis and Generation of Sentences," *IBM Journal of Research and Development*, Vol. 32, No. 2, March, 1988, pp. 251-267.

[109] Vogt, F., "Formal Concept Analysis as a Means for Analyzing and Exploring Data," *Computer Science Department Technical Seminar*, The University of Alabama in Huntsville, Computer Science Department, March 19, 1997.

[110] Waters, R.C., "A Method for Analyzing Loop Programs," *IEEE Transactions on Software*

Engineering, Vol. SE-5, No. 3, May, 1979, pp. 97-106.

[111] Watson, M., *Portable GUI Development with C++*, McGraw-Hill, 1993.

[112] Weiskamp, K., and Flamig, B., *The Complete C++ Primer*, Academic Press, Cambridge, MA, 1992.

[113] Winograd, T., *Language as a Cognitive Process, Volume I: Syntax*, Addison-Wesley, 1983.

[114] Yourdon, E.N., and Constantine, L.L, *Structured Design*, Yourdon Press, 1978.

[115] Zarri, G.P., "Automatic Representation of the Semantic Relationship Corresponding to a French Surface Expression," *Proceedings of the Conference on Applied Natural Language Processing*, Santa Monica, CA, 1983, pp. 143-147.