A consignment library of reusable software components for use over the World-Wide Web

by

R. Austin Hicklin

Research project report submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

APPROVED:

Mary Beth Rosson, Chair

J.A.N. Lee

Edward A. Fox

May 24, 1995

Blacksburg, Virginia

Keywords: Software Reuse, Reusable Software Libraries (RSLs), Software Classification, Reuse Incentives, Digital Libraries, World-Wide Web (WWW)

A CONSIGNMENT LIBRARY OF REUSABLE SOFTWARE COMPONENTS FOR USE OVER THE WORLD-WIDE WEB

by

R. Austin Hicklin

Mary Beth Rosson, Chairman

Computer Science Department

This research project report discusses the development of a commercial, consignment-based library[a] of reusable software components to be accessed using the World-Wide Web. The research project consists of two parts: the development of a prototype system that provides interface and information retrieval functionality for such a system, and an analysis of the technical and business issues involved in making the library operational as a commercial entity.

The prototype system uses a hypertext browser and a query-based search mechanism to access descriptions of reusable software components; these descriptions are structured by a variation of a faceted classification system. The issues addressed include the classification and description of reusable software components; methods of retrieval, especially library browsing methods based on component classification; and analysis of incentives for reuse.

---

a *The expression "commercial library" is admittedly somewhat self-contradictory, but usage of the phrase "digital library" has come to be applied generally, to both commercial and public enterprises.*

TABLE OF CONTENTS

List of Illustrations

List of Tables

1 PROBLEM DESCRIPTION

The software crisis first described in 1968 [Naur68] is not over, even though the software development process is maturing, and great advances have been made in software engineering. Software reuse has been promoted as a method of bringing order to the process of software development, and has seen some success, but the lack of large, generally available repositories of reusable software components[a] is an impediment to large-scale software reuse.

Two problems with the implementation of large repositories of software components have to do with the accessibility of the library, and with the incentives for building and maintaining the library. Access to software reuse libraries has been inconvenient in the past, limited by such means of access as paper catalogs or email messages; the popularization of the World-Wide Web provides a solution to the problem of accessibility in that it provides a standard, widely used interface that can be used in conjunction with powerful information retrieval tools. Incentives for the expensive process of building and maintaining a library have been lacking in the past. I believe that a consignment[b] method of sale and purchase may provide a solution to the problem of incentives by providing an economic benefit for each of the three parties involved in the process of reuse: the *component developer* would be able to sell software components that might not otherwise have a market; the *library* would be able to charge a fee for each component sold; the *component user* would be able to reuse components that might not otherwise be available.

A World-Wide Web site that would provide a central location for the sale and purchase of software code by consignment may be of benefit to the computer science

---

[a] *There is some disagreement over the use of the term "component", but generally, it means any type of software that can be reused, including code fragments, classes, and compiled objects.*

[b] *In a consignment system, developers are not paid when their components are added to the library, but receive payment (minus a percentage or fee for library overhead) when the components are sold.*

community by encouraging large-scale code reuse, and may in addition have the potential of being a successful business. This report describes the development of a prototype system that provides interface and information retrieval functionality for such a library, and analyzes the issues involved in making the library operational as a commercial entity.

There is good reason to believe that such commercial libraries are likely to become very important in the near future. Since there is an economic benefit present for all parties involved, there is a practical opportunity for the creation of a new marketplace. If one or more libraries of this type becomes established on a very large scale, then the process of software development may change in reaction: languages and development environments may change because they can rely on the presence of large numbers of reusable components; a class of professional freelance component developers could be created since they would have a reliable market for their products; the process of software development would be simplified, since development would be pointed more toward the software integrated circuits hoped for in the early days of reuse {McIlroy69].

This document is organized in the following manner: this section (Section 1) provides a brief overview of the problem, Section 2 describes the Component Library as implemented, Section 3 analyzes a variety of issues in light of previous work and design decisions made during the development of the prototype system, and Section 4 suggests directions for future work.

2 PROJECT DESCRIPTION

2.1 Background

Reuse is based on an accretion of knowledge and experience; it is an active application of historically acquired understanding and artifacts. All communities

possess a communal memory; civilized society is itself a repository of knowledge, in that civilization and technology are built upon a common virtual library of societal understanding, experience, and physical artifacts.

In the field of computer science, we have always made great use of such a virtual historical library of understanding and experience, but have been slow to adapt to the reuse of actual code: we reuse the concepts and algorithms on which code is based, but the actual code, with slight variations, is written and rewritten *ad nauseum*. Reuse is an especially intriguing concept in software development (and its limitations are particularly galling) because in few areas of human endeavor is the exact duplication of an object such a trivial effort.

Large-scale software reuse has been an elusive goal of software engineering for some time. McIlroy first proposed the concept of the reusable software component in 1968 [McIlroy69], and although a number of years passed before much progress was made in this area, a variety of recent successes have made the proponents of large-scale reuse more sanguine. These developments include the increasing acceptance of object-oriented languages and abstraction methods, availability of some libraries of object-oriented classes (such as Smalltalk or C++ class libraries), domain analysis advancements [Neighbors94], and management incentives increasing the rate of reuse [Frakes94b]. Of particular interest here, however, are the advancement of digital library technology and the successful commercial development of the Visual Basic custom control market.[a] Digital libraries, in conjunction with advances in information retrieval methods and widespread interconnectivity, have made it possible to create and access very large dynamic libraries of information. Visual Basic custom controls have created the most commercially successful market yet seen in reuse, and the development of OLE 2.0, OpenDoc and other application- and (theoretically) platform-independent component standards anticipate the development of a stable, widely accepted

---

[a] *VBX components for Microsoft's Visual Basic can be used in Visual C++, Borland's Delphi, and some other development environments. OLE 2.0 components (the most recent custom control format from Microsoft) can – at least to some extent – be accessed via a number of applications, including Microsoft Word and Excel, and are intended to work on Macintosh equipment. [Udell94]*

community of reuse [Udell94]. Application and platform independence are extremely important advances in reuse: a fundamental complication for reuse has been the dependency of components on language and operating system requirements.

Two of the primary obstacles to the implementation of large-scale software reuse libraries are logistics and the lack of viable incentives for their growth. In the past, learning what components were available and retrieving them were both limited by the means of distribution: sophisticated searches of the library and quick retrieval were not possible through paper catalogs; specialized library software could provide these benefits, but would not be universally distributed. The growth of the World-Wide Web has introduced a new alternative: libraries can be accessed using generally available browsers, which can serve as front ends to sophisticated information retrieval methods, and components can be downloaded directly. The World-Wide Web means that implementing a universally accessible library has become fully practical. Indeed, over the past twelve months, several projects have been started that are attempting to build reuse libraries for use over the World-Wide Web (see Section 3.4.2); none of these projects, however, is building a generally accessible commercial library, as they are either focusing on components of a research nature, or are dealing with proprietary or secure information.

Most research ignores or finesses any impetus for the creation of large-scale software reuse libraries: private companies are working on large reuse libraries for internal access, but this does not further the cause of industry-wide reuse. A general assumption is that federal funding is necessary to build the infrastructure for such libraries. For any specific market to be successful in our economy, economic incentives must encourage both supply and demand in that market. For libraries of reusable software components to thrive, there must be economic incentives to provide components for the library, to maintain the library, and to use the components. Existing reuse libraries are currently constrained in size because there is little impetus for growth (other than federal funding): code may be written specifically for the library, which is extremely expensive; the libraries may be composed of public-domain software, which does not encourage developers to add to the library; the libraries may

be composed of shareware, which provides an uncertain prospect for income for developers.

To the best of my knowledge, the work presented in this document is unique in its attempt to build a commercial, consignment-based library of reusable software components that will be accessed via the World-Wide Web.

2.2 Approach

For this project, a prototype system (referred to in this document as the "Component Library") has been completed that will act as a working basis for a commercial component library, and a preliminary analysis of the issues involved in building such a library has been conducted. Future research is planned that will address the issues raised in this analysis and build upon the Component Library to create a fully operational commercial component library.

THE PROTOTYPE SYSTEM

The Component Library is an interface to a collection of descriptions of software components — note that for this project, the "library" consists of the descriptions of the components, not the components themselves. This system allows a user to access the library with a HTML+ compatible Web browser.[a] Abstracts for the components can be accessed through a browsing and querying system. When the user selects a component, a complete description of the component and purchase information are displayed. If the user chooses to purchase the component, the system prompts the user for a credit card number, and the component is transferred back to the user through the user's Web browser.[b] Complete scenarios of how the system might be used are presented in Section 2.3.

---

[a] *HTML+ is the designation for the proposed HTML 3.0 standard implemented by Netscape version 1.1; see the following section for an explanation of this requirement.*

[b] *The issues of security, verification, and other payment options are discussed below. Files are transferred via FTP (file transfer protocol). Since the components themselves are not present in the library, ftp is not enabled in the prototype system.*

IMPLEMENTATION

The Component Library was developed on an Ultrix-based DEC 5000 owned by the WebStation Company, running a NCSA HTTPD 1.3 World-Wide Web server. The programs that make up the Component Library are written in Perl, and are accessed as Common Gateway Interface scripts through the Web server. Perl was used because its text processing and Unix command processing features are more powerful and convenient than those features of Unix shell scripts or C, and because the fact that it is interpreted rather than compiled makes it more convenient for use in prototyping than C. Note that all of the pages in the Component Library are generated by Perl scripts; none are HTML documents. The use of scripts in this way allows communication of information between pages, which is not possible in static HTML documents. freeWAIS version 0.3 was used to process queries,[a] but for the browser it was decided that WAIS was not an appropriate tool, since the browser requires a deterministic retrieval of information, while WAIS returns information using a probabilistic method. The browser was implemented using a set of inverted files; using different files for different sections of the components' classifications was found to be most effective in terms of response time.

The use of the World-Wide Web as an access mechanism introduced some limitations in the design of the user interface. Early HTML versions included a very simple set of markup commands; forms were implemented in HTML 2.0, and tables are to implemented in HTML 3.0 (the definition of which has not been finalized). The use

---

[a] *WAIS is a system originally developed by Thinking Machines Corporation, based on the ANSI/NISO Z39.50 protocol. Thinking Machines Corporation defined a set of extensions to the Z39.50 protocol and labeled the resulting standard Z39.50-1988. A variety of WAIS clients and servers were developed based on this standard; commercial WAIS versions were developed by WAIS Incorporated, while public-domain WAIS versions were named freeWAIS and made available through CNIDR. freeWAIS versions based on the Z39.50-1988 standard were designated with a version number of 0.x; freeWAIS 0.3 appears to be the most robust version of this series (we were not able to install freeWAIS 0.4 on our Ultrix platform). CNIDR also has WAIS versions based on the Z39.50-1992 standard, which has greater functionality than, but is not compatible with, the Z39.50-1988 standard; public-domain versions of WAIS based on the Z39.50-1992 standard, once known as freeWAIS 1.x, are now designated as ZDIST 1.x. [CNIDR95] [freeWAIS]*

of tables allows a much more effective use of screen space, so it was decided to require HTML+, which is Netscape's designation for the proposed HTML 3.0 standard. Although requiring the use of Netscape 1.1 as a browser is a limitation at the moment, it is assumed that when the HTML 3.0 standard is accepted, this functionality should be implemented in most Web browsers [Netscape].

The use of HTML+ allows a certain degree of graphical design control not available in HTML 2.0, in that font sizes can be specified and text can be justified. Although for the design of text documents it is generally conceded that descriptive markup systems should be used instead of presentational markup [Coombs87], Coombs's evidence is limited to the linear display of "scholarly" text, and should not be taken to define methods of display for other forms of text, or for software interfaces; for this reason, it was decided to use the presentational markup facilities available through HTML+.

The presentation of information in a Web browser demands careful consideration of the use of screen space; informal Web document style suggests that pages be limited whenever possible to a few screens. For this reason, it was decided to use tables and pulldown menus in the Component Library: the main interaction page fits on a single screen, and no page is longer than two screens.

The use of pulldown menus has a drawback in interactivity: while a user selection of a hypertext anchor in a Web browser is communicated immediately to the server, a user selection in a menu is not relayed to the server until the user chooses to submit the form's information. For this reason, the system cannot immediately respond to user selections, even though such feedback would be desirable. Preliminary evaluation has shown that while the interactive feedback gained by displaying options as text (rather than in menus) is useful, the resulting demands on screen space were unacceptable.

EVALUATION OF THE PROTOTYPE SYSTEM

A variety of methods of implementing the prototype system were considered, and several were in fact implemented, including pure hypertext browsers and several methods of faceted classification. These early designs were evaluated informally by six

other computer science graduate students; evaluations were conducted on scenarios of use, on screen mock-ups, and on working systems. The library as implemented was preferred by the evaluators to the options presented; the reason overwhelmingly mentioned by the evaluators was the simplicity of the interface as implemented. The user population will be software developers, and will therefore be moderately to extremely sophisticated in technical matters; nevertheless, research has shown that complex library interfaces degrade usability regardless of the user's technical abilities [Poulin95a], so simplicity of interface is of primary importance.

2.3 Scenarios of Use

To illustrate the use of the prototype system, two scenarios are outlined here; the design of the system and the rationale for the implementation decisions will be explained in Section 3.

SCENARIO # 1 (BROWSING)

Sue is in the process of developing a SQL database application in Visual Basic, and wants to know what utilities are available for her to use. She follows a link to the Component Library and gets the Component Library's main screen (see Figure 1). The following information is presented to her:

- The system tells her how many components are in the library, and provides a link to display the entire list.
- Selections can be made in the following categories: *Type* (see Figure 2), *Language* (programming language, for source code components), *Operating System, General Purpose* (see Figure 3), or *Specific Purpose.*
- Search terms can be entered (see Figure 10).
- After terms have been selected or entered, a button must be pressed to submit the selection criteria.

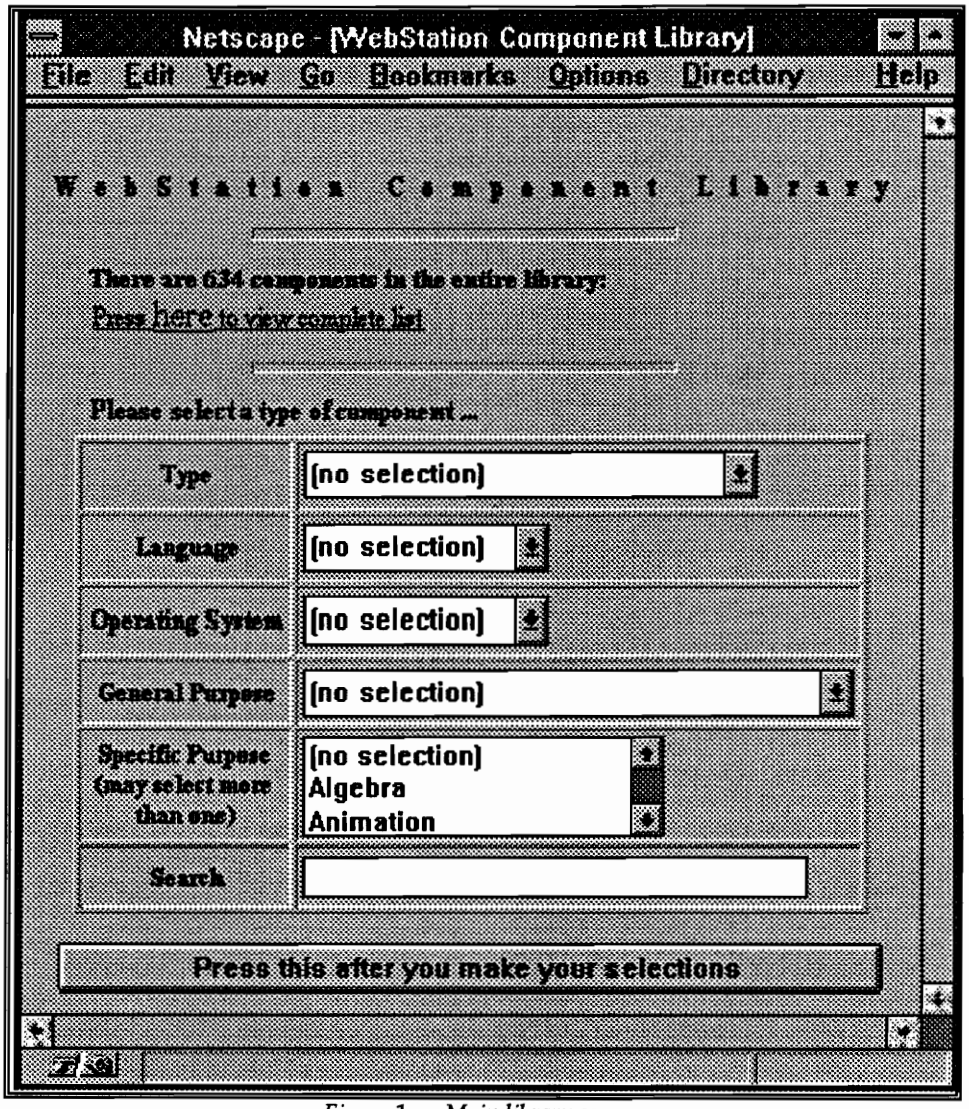

Figure 1 – Main library screen

Since Sue is only interested in VBX components, she selects *VBX component* as the component's *Type* (Figure 2).

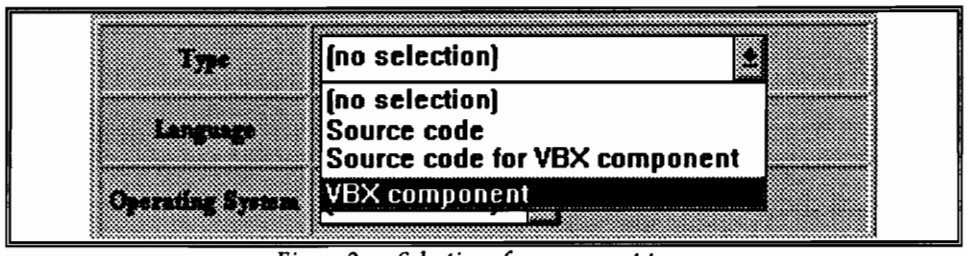

Figure 2 – Selection of a component type

Sue selects *Database* as the component's *General Purpose* (Figure 3).

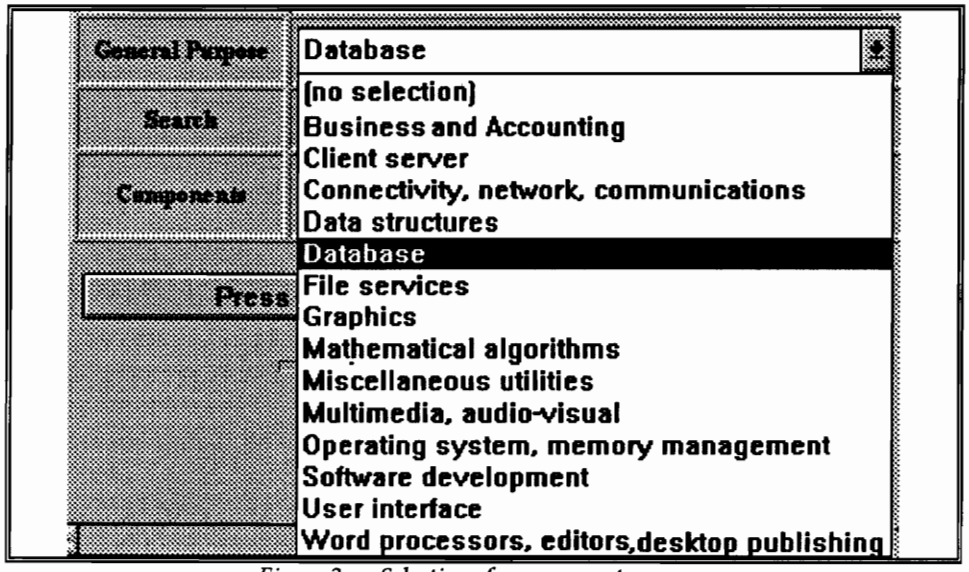

Figure 3 – Selection of a component purpose

Sue submits her criteria by pressing the button labelled *Press this after you make your selections* and the screen updates (Figures 4 and 5 are on the same Web page, but had to be split for printing):

- She is shown a summary of her selections and an alphabetical list of the components in the library that meet those selections. Since she has made selections, the list of applicable components has been limited to 12; when the list of components is reduced to 40 or fewer, they are displayed on the response page (Figure 4). If the list contains more than 40 components, they can still be viewed, but are not displayed on the main page, since displaying such potentially long lists can add substantially to the time needed to display Web pages. (see Figure 1).

- Operating System is set to Windows 3.1, since the VBX components in the library all operate under Windows 3.1.
- Likewise, if she looked in the Language field, she would find C and C++ as the only remaining options, since VBX components must be written in C or C++.
- The Specific Purpose field lists all keywords that occur in conjunction with the General Purpose she selected earlier.
- A button to reset selections has been displayed.

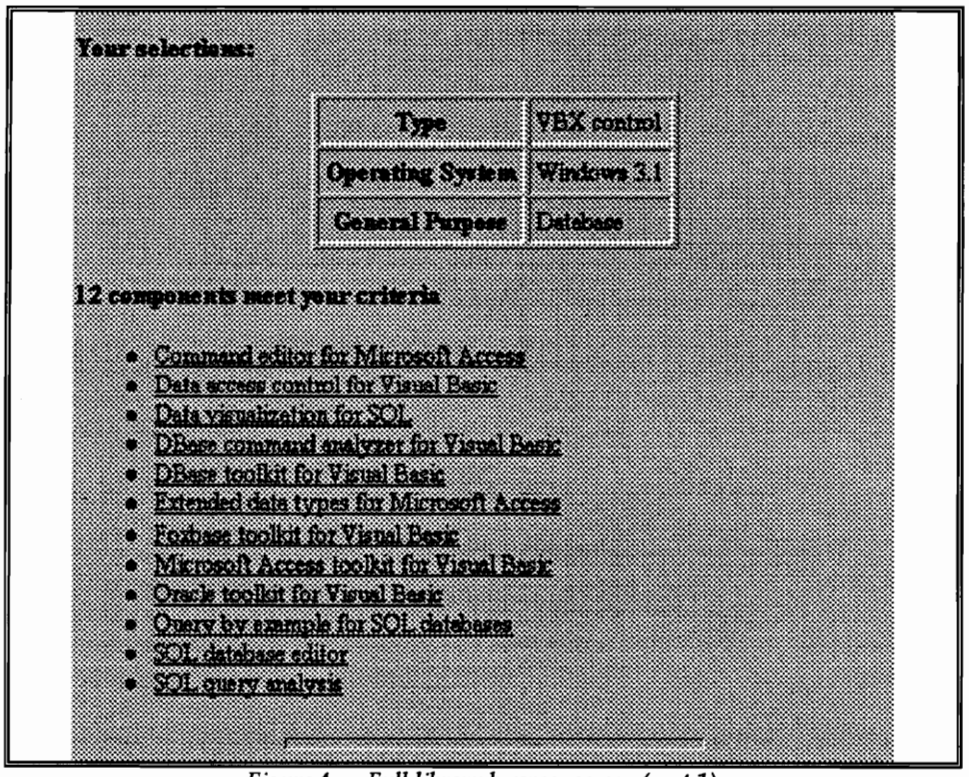

Figure 4 – Full library browser screen (part 1)

You may further restrict your selections :

Type	VBX components
Language	(no selection)
Operating System	Windows 3.1
General Purpose	Database
Specific Purpose (may select more than one)	(no selection) Charts Data display
Search	

Press this after you make your selections

Press this to reset selections

Figure 5 – Full library browser screen (part 2)

Sue selects the component named *SQL Database Editor*.

A complete description of the component is displayed (Figures 6 and 7). Sue browses through several component descriptions and prints them out for later reference. She is particularly interested in the *SQL Database Editor*, so she decides to buy it; she selects the *Purchase this component* option.

WebStation Component Library

Purchase this component

Return to the browser

Name	SQL Database Editor
Overview	Spreadsheet-style interface used to access and update SQL databases
Component purpose	Database, Business, Accounting, Spreadsheets
Keywords	Database utilities
Component type	VBX control, Source code
Programming language	C++
Operating system	Windows 3.1
Description	The SQL database editor is a Visual Basic / Visual C++ component that allows access to any fields on a database. The data is displayed in a spreadsheet-style grid. Two keys are selected, which are displayed on the vertical and horizontal axes; the fields are displayed on the grid corresponding to the appropriate keys. Two key modes are available: the application can set the keys to be used, or the user can select the desired keys within the component. Two update modes are available: view only, and view / update. In update mode, the database is modified directly, and the component does not control or monitor the update.

Figure 6 – Component description (part 1)

Version	1.0
Number of copies sold	23
Price	$200 ($400 with source code)
Input	Database name, horizontal key, vertical key properties
Output	Directly modifies database (can be set to view only)
Quality assurance	Programmer tested; thoroughly documented code, test data and testing methods available from developer on request; money back guarantee
Documentation	Readme file (60K), help file; 3 sample programs
Developer ID	D-1867
Copyright restrictions	All rights reserved
Comments	Limited money back guarantee from developer if not satisfied! OLE 2.0 version under development.
Component ID	vbx_sql_edit
File name	sql_edit.zip
Compression method	PKZip v. 2.0
Date released	3 Nov. 1994
Memory requirements	640K minimum; 2M desirable
Size	230876 bytes, 76256 bytes compressed

Figure 7 – Component description (part 2)

A purchase form is displayed (Figure 8). She enters the requested information and submits the information.

WebStation Component Library

Name	Sue
Email address (must be on the machine you are browsing from)	
Physical address	
Telephone number	
Credit card type	Visa: Mastercard: Discover: American Express:
Credit card number	
Credit card expiration date	Month: 1 Year: 95

Press this after you have entered all information

Copyright © the WebStation Company, 1995

Figure 8 – Purchase form

The prototype system contains only component descriptions, not the components themselves, so the existing system simply echoes the purchase information. If Sue were using the fully implemented system, then after her credit card was verified, Sue would be prompted for a local file and directory in which to save the component (Figure 9); if she selected *OK*, the file would be transferred.[a]

[a] *If she cancels the transfer or the file transfer is not completed successfully, the credit card transaction would be cancelled. Since a user could download most of a component and purposely cancel the transaction to get most of the code without paying for it, the components should be compressed with an encryption password.*

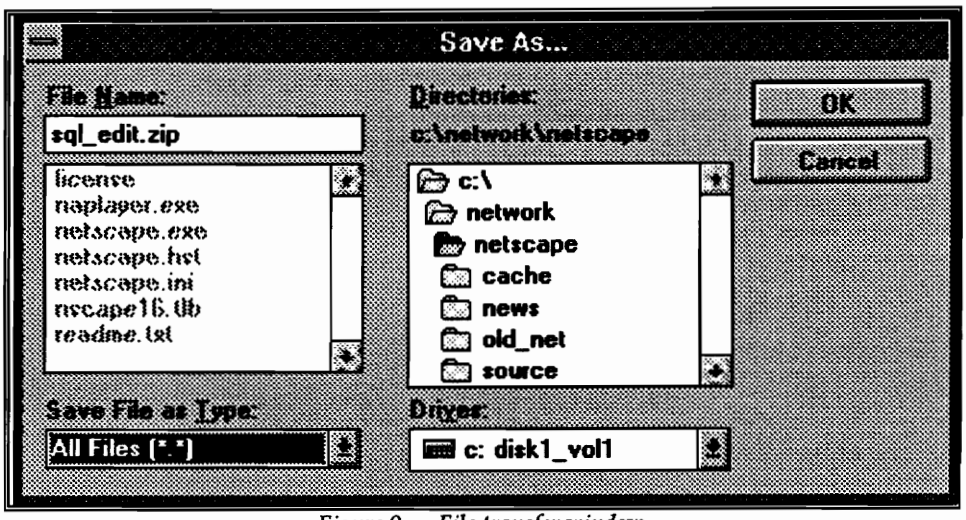

Figure 9 – File transfer window

SCENARIO # 2 (SEARCHING)

Fred also wants to find VBX database components. However, instead of browsing through categories from the main screen (Figure 1), he enters *VBX database* in the *Search* field (Figure 10).

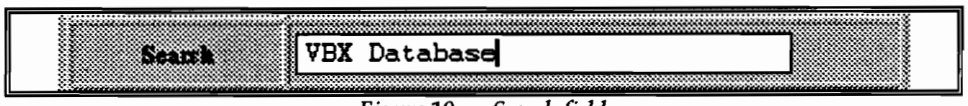

Figure 10 – Search field

In response to his query, the system returns the components as a ranked list (Figure 11). This query is used to search both the components' descriptions and their classification information (the attributes of the components such as the component type, programming language, and pupose), while the browser selections are based only on classification information, so the results of a query may be broader than the results of an equivalent browser selection. Query and browser selections can be used in conjunction with each other: queries can be used to limit the list of components returned in response to a browser selection, and browser selections can be used to limit the list of components returned in response to a query.

The rest of Fred's interaction with the Component Library would be identical to Sue's.

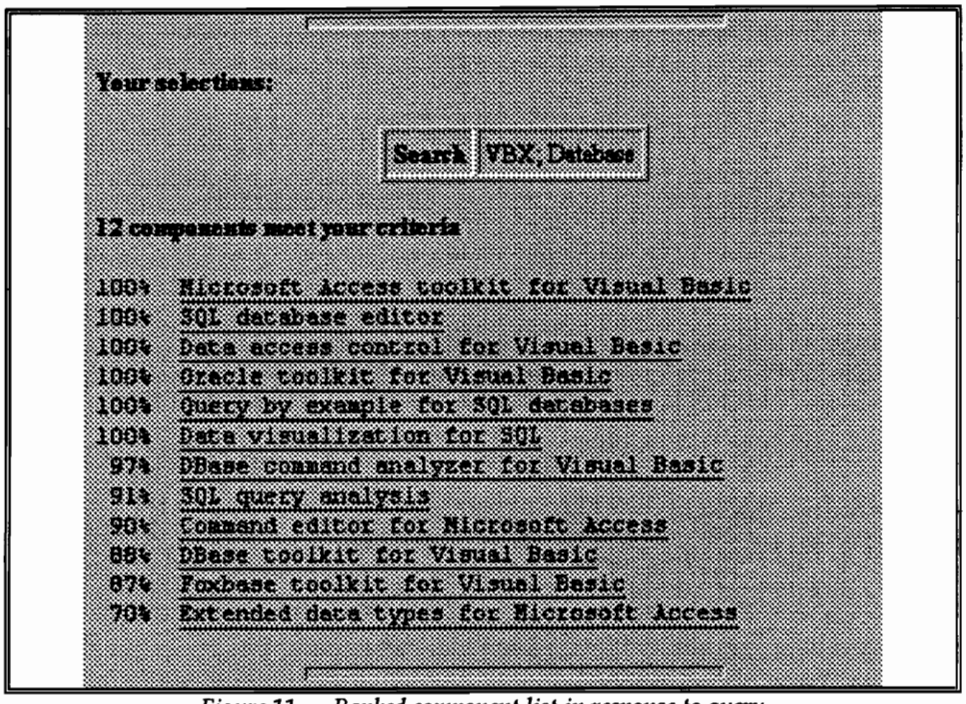

Figure 11 – Ranked component list in response to query

3 REVIEW AND ANALYSIS OF ISSUES

During the design and implementation of the Component Library, a number of research issues have been addressed. In the following sections, previous work relevant to each of these issues is reviewed, and design decisions in the current implementation are explained. The following issues are covered in depth in Sections 3.1 - 3.4.

Reuse ..... What components can and should be reused, and how do their differences affect the library?

Classification and Description How should components be classified to facilitate both retrieval and understanding? How can an understanding of the workings of software components best be imparted?

Browsing and Querying ..... How can users communicate with the library to determine which components best suit their needs? How can browsing and querying be designed to be most useful and usable?

Scaling.....How is the usability of the library affected by the number of components in the library? How can the retrieval mechanism be designed to be most useful for large libraries?

3.1 Reuse Considerations

A certain amount of reuse has always been present in software development: taking related routines or programs and making ad hoc modifications to them is a standard programming practice. Although in some circumstances ad hoc code reuse ("code scavenging") may be appropriate and necessary, it can be both inefficient and error prone, and tends to be limited to localized development; research into more formal methods of reuse is attempting to find systematic approaches that are more elegant, efficient, reliable, and universal. It may be argued that software reuse is the next logical step in the progression of software development in terms of abstraction: information hiding, or the encapsulation of complexity, has moved through the abstractions of procedures, modules, and abstract data types; the use of local and universal component libraries would be logical additions to that progression [Krueger92].[a]

Attempts to create a field of systematic software reuse date back to the 1968 NATO Software Engineering Conference, at which the entire field of software engineering is said to have had its origins [Krueger92]. At that conference, M.D. McIlroy outlined his view of the creation of a "software components subindustry," through which developers would have a truly huge selection of components that could be used as building blocks in the construction of applications. These components would not be designed to be general enough for all purposes; instead they would be available in a wide variety of "models" so that specific needs of, e.g., precision or time-space requirements would be exactly met without any tradeoff in efficiency [McIlroy69].

---

[a] *Krueger has argued that the language constructs in high-level languages are themselves examples of code reuse [Krueger92].*

According to Biggerstaff and Richter, software reuse research is generally seen to address four major issues [Biggerstaff89]:

- Components must be *accessible*; developers who wish to use software components must be made aware of what is available, must be able to determine how the available components relate to their needs, and must be able to acquire the components.
- Components must be *understandable*; components must be described or documented in ways so that developers who use them can recognize the implications of their use.
- Components must be *modifiable*; in some cases, the component must be adapted for its new use.
- Components must be *composable*; the receiving system must be made able to coordinate with a new component, and components must be able to work together.

This classification does not note that components must be *created*; this may involve automatic generation of components, processing of existing code to segregate autonomous components, or design and writing of code specifically for use as Component.

Of these issues, the problems of access and understanding are most important in the development of reuse libraries. The modification and composition of components are of secondary importance here, except in the way that they relate to understanding of the components; the creation of components must be addressed to some degree insofar as determining what components should be included in the library and how they are to be acquired. The primary reuse-related issues to be addressed in the development of component libraries are

- defining what is to be reused;
- addressing methods of access, particularly in terms of browsing and searching methods; and
- addressing issues of understanding, and the methods by which these can be solved by description, abstraction, and classification of the components.

3.1.1 Types of Reusable Components

McIlroy's view of software components was limited by his era. In the late 1960s methods of encapsulation were limited:[a] components had to be pieces of programs, whether fragments or functions or data structures. Reuse of unencapsulated code is still very prevalent, whether in libraries of mathematical algorithms [GAMS94], in code automatically extracted from existing systems [Caldiera91], or in ad hoc code scavenging. Since that time, a great deal of attention has been paid to object-oriented design and analysis, which has greatly affected how reusable components are defined.

One of the primary purposes of object-oriented design is to facilitate reuse. Of the basic tenets of object-oriented design, encapsulation, inheritance, and message passing all affect the ways that components can be formed.[b] Encapsulation makes general-purpose reuse practical: components can be designed so that they are nearly self-sufficient and can be incorporated into other systems without any requiring any modification to the component. Inheritance allows a component to use another component as a basis so that only code specific to the differences between them needs to be written. Message passing allows composition of components in a straightforward but controlled manner.

In the more than twenty years since Smalltalk was developed, however, object-oriented technology has not enjoyed unqualified success. Smalltalk is growing in use in commercial environments, but is still a minor player. C++ has become very important in commercial use, but its own inventor, Bjarne Stroustrup, has "admitted that the appropriate scope for reuse of C++ modules was probably the project or department level" [Udell94]; the strict typing of C++ means that applications usually have to be recompiled when component classes are changed, and that linking is not compatible from compiler to compiler [Udell94].

---

[a] *Algol was the most advanced language in this direction at the time; McIlroy criticizes Algol as inappropriate for many applications.*

[b] *Polymorphism has little or no effect on the formation of components; some would argue that message passing is not a fundamental tenet of the object-oriented method.*

DYNAMIC LINK LIBRARIES AND VISUAL BASIC CUSTOM CONTROLS

These compilation and linking problems are not present in the Dynamic Link Libraries (DLLs) for Microsoft Windows. Programs in Microsoft Windows can store common routines in such libraries and resolve references at run time. These libraries were designed for the purpose of reuse, but are object-oriented only in terms of encapsulation and messaging [Petzold92], not inheritance or polymorphism. Variations of DLL known as Visual Basic custom controls (VBX) were designed to be manipulated by Visual Basic, which functions essentially as glue to connect the components. Components communicate with the calling application through the passing of events (such as keypress, mousedrag, etc.) or through an array of properties (which is used to change or display the current value of component-specific values such as title, location, etc.). This standard interface to the calling program gives DLL or VBX components a portabililty that is not present in most other types of components.

The implementation of DLL or VBX components certainly has its limitations, but DLL and VBX have succeeded because they have made the reuse of components truly trivial. Object Linking & Embedding (OLE 2.0), the successor to VBX controls, has added much greater functionality, including inheritance, application independence, and (supposedly) platform independence; other component types, such as Apple's OpenDoc, are moving in the same direction. Note that some responsibility for the success of Visual Basic controls may lie in the increased cognitive distance between the users' and developers' perspectives of Microsoft Windows: while *use* of graphical user interfaces instead of text-based interfaces is much easier, *development* for such interfaces is much more complex; Visual Basic attempts to bridge this gap between users and developers. Since much of the code required for GUIs is relatively complicated, but highly redundant [Biggerstaff89], it is particularly well-suited for reuse. Visual Basic has carved out a niche for itself based on its interface and database components; whether this share is likely to grow is debatable, but since a great deal of commercial software is dominated by interface and database code, it seems likely that Visual Basic and its descendants will continue to increase in importance well into the future [Udell94].

WHITE-BOX AND BLACK-BOX REUSE

Visual Basic components are compiled in the form of linkable libraries and are linked into an application, without any need of knowledge on the programmer's part of how they are constructed; this is called black-box reuse. White-box reuse involves knowledge of the inner workings of the component, either so that the component can be modified for use, or for verification and quality assurance purposes.[a] The distinctions between black- and white-box components have great implications for reuse. In brief (see Section 4), black-box components have quality assurance problems, since the reliability of the components must be taken with a certain degree of faith; white-box components have greater description requirements, since a certain level of understanding of the component is necessary for reuse [Neighbors94].

GRANULARITY OF REUSE

The size of the components stored in a library can have implications beyond that of simple storage space. In large grain reuse, the individual components are large and therefore (presumably) complex to describe; in small grain reuse, the individual components are small and therefore easier to describe, but if the decrease in individual size translates into an increase in the total number of components in the library, library management problems become more serious.

3.1.2 Reuse Considerations in the Component Library

Reuse considerations, obviously, affect the entire project. The decisions made specific to this area are primarily a determination of what types of components to use when populating the library. As shown above, the type (black box or white box) and granularity of components have implications for several aspects of library development. In addition, the language and platform of the components in the library have a great impact on who will access the library: for example, Visual Basic components will be of interest primarily to business developers and hobbyists, FORTRAN routines will be of

---

[a] *There is some disagreement as to whether this last case is actually white-box reuse.*

greatest interest to mathematicians, and Smalltalk objects will be used mostly by academics (but by some business users).

Much of the decision regarding the components to be used in the library is driven by availability. As a commercial concern, the library should take advantage of the components made available by developers regardless of type. The library should focus its efforts, however, on those areas most likely to be commercially viable. With the current state of the industry, the greatest interest in commercial reuse appears to be in Visual Basic components (primarily OLE 2.0, with lessening interest in VBX and DLL controls), and C++ classes; classification and browsing in the Component Library have been optimized for the particular needs of these types of components. There may be certain difficulties in stocking C++ code, due to dependencies on specific libraries (e.g. Borland C++ libraries), or on platforms (e.g. MS Windows or UNIX); these concerns can be worked around, but add description and classification difficulties.

3.2 Classification and Description

A description of a component serves two different purposes: it must convey a sufficiently detailed understanding of the component, and it must provide an overview so that it is easy to tell whether a component is of interest. The process of description is a process of abstraction, in that the complexity of the original form of the component is replaced by a simpler form, in which only information deemed to be relevant is retained; like any form of abstraction, it results in the hiding of information. Information hiding in this case has both positive and negative aspects: positive in that the component is represented in a form more likely to be readily understood, but negative in that valuable information might not be represented in the abstract form, or might not be represented correctly.

The detailed description of the component must attempt to convey all necessary information to the prospective user: for a component to be selected for reuse, its purpose must be explained; for it to be reused effectively, the component's interface to the receiving system and the effects of all possible variations of input must be explained. In addition, the type of the component has an effect. White-box components

must be explained exhaustively, since the implementation will be modified during integration of the component into the receiving system; in non-commercial libraries, the source code may be regarded as part of that documentation, whereas for commercial libraries the fact that the source code is not available until after purchase must be taken into consideration. For black-box components, a description of internal workings is not necessary.

The quality and detail of component descriptions vary greatly [Sørumgård93]. If descriptions are written by the individual developers, who frequently have a poor reputation for documenting code, presumably have limited interest in writing such descriptions, and whose writing skills vary from good to very bad, then the quality of the component descriptions will be erratic. If, however, the descriptions are written by professional writers, a certain uniformity and quality of writing is brought to the descriptions, but the descriptions may be inaccurate, since the writers cannot be expected to have an intimate knowledge of the code. A compromise might be to have developers provide content, using a template, that would then be edited by a technical writer.

Classification is a substantial part of description. Appropriate classification will allow users to browse through component descriptions effectively, and may be responsible for the user's determination of whether or not a component is relevant. Very small collections of components need little in the way of classification, since the entire set of components can be browsed to determine their possible uses. But as the size of the library grows, the need to distinguish rapidly between components increases, and the classification system must become more discriminating and robust [Sørumgård93].

3.2.1 Review of Classification Schemes

Classification schemes are generally broken into keyword, enumerative, and faceted schemes. In free text keyword schemes, an unstructured list of descriptors is

generated by the contributor for classification purposes. This has the advantage of being very simple from the point of view of the classifier, and it allows the classifier to choose exactly the terms desired. Unfortunately, since people frequently choose different terms to represent a given concept [Furnas87], classifications developed in this way may mean different things to different people.

ENUMERATIVE CLASSIFICATION

In enumerative classification, components are placed in a tree: a strict one-dimensional hierarchy of categories, in which each component is a member of a single elemental class, which along with other elemental classes is a member of a superclass, and so on [Prieto-Díaz91] [Sørumgård93]. An example of enumerative classification is that used by GAMS — the Guide to Available Mathematical Software (see Section 3.3.1). GAMS uses a method based on a software classification scheme developed in the 1960s by SHARE, the IBM users group, and reached its current state in 1983. This taxonomy classifies approximately 9000 mathematical algorithms in a tree with 736 nodes, 21 main classes and a depth of up to 8 levels [Boisvert94]. For example, a routine to apply quicksort to an array of integers would be classified in this way [GAMS94]:

Table 1 — Sample GAMS classification

• N. Data handling (search also class L2)
• N6. Sorting
• N6a. Internal
• N6a2. Active
• N6a2a. Integer

Note that the classification makes no mention of the actual algorithm, the data structures, the language, or the environment: these are included as attributes of the component, not parts of the classification. Note also the "(search also class L2)" comment — the hierarchy is not capable of removing ambiguity or overlap.

Enumerative classification is effective in GAMS because mathematical and computational routines are a special case in component classification. The domain of mathematical algorithms is very well understood and defined, and is evolving so slowly that it is nearly static. In general, enumerative classification is difficult to expand except in detail; for mathematical algorithms, however, the domain is mature and (very)

unlikely to change in structure, so the rigid enumerative scheme can be developed with a holistic understanding of the overall domain [Biggerstaff89]. Enumerative classification is difficult to maintain for areas such as general-purpose software that are still in a constant state of change, since the structure of enumerative schemes cannot easily be expanded or refactored. Enumerative classification is particularly unsuited to domains in which some subclassifications overlap [Prieto-Díaz91] [Sørumgård93].

ATTRIBUTE-VALUE AND FACETED CLASSIFICATION

For domains that are incompletely defined, or that must be described by multiple overlapping aspects, enumerative schemes are not as appropriate as attribute-value or faceted schemes. Attribute-value and faceted classification both define a component in terms of a number of attributes, to each of which a value is assigned. The distinctions between attribute-value and faceted classification are subtle (and the authorities do not agree), but in general, they are distinguished in the following ways:

- Faceted classification limits the possible values (termspace) to a finite list [Poulin93].
- Facets are generally chosen to be as orthogonal as possible to each other [Sørumgård93].
- Components are usually described by using no more than seven facets [Frakes94c].
- The order in which facets and values are presented is relevant [Frakes94c].
- Processing of synonyms is generally provided in faceted schemes [Frakes94c].

First described by Ranghanathan in 1957 [Ranghanathan57], faceted classification looks at a domain from two or more fully or partially orthogonal perspectives or dimensions. Each of these facets is in itself an enumerative classification scheme; the entire faceted system is a synthesis of these enumerative classifications [Sørumgård93]. An example of a faceted scheme could be based on the example of the enumerative classification used above: if a component has facets of *algorithm, language,* and *platform,* then a UNIX-based C++ Quicksort routine might be classified as *(algorithm: Quicksort; language: C++; platform: UNIX).* Clearly this method of

classification is much more flexible than an enumerated method, so it is particularly appropriate for domains that are in a state of flux, and which therefore demand changes in classification structure. It is important to note that any entity may be classified by either system [Prieto-Díaz91]. In an enumerative scheme, all possible classifications must be explicitly listed, which is both an advantage and a disadvantage. If it is important that a domain be very well defined, a faceted scheme may not be appropriate.[a]

How the categories and values of facets are determined is a matter of some dispute. Sørumgård states that "information categories (facets) should be well defined and as orthogonal to the other facets as possible." [Sørumgård93] One difficulty with this perspective is that at times a faceted breakdown may be useful within a facet category, which is not an orthogonal organization. The benefits of a system that allows facets within facets may be that complexity is hidden in this way, and that the structure can be made more representative of the user's understanding of the domain; the drawbacks may be that this adds complexity to the system, and that users may not view the domain in terms of facets [Poulin95b]. Complexity in a classification system is counterproductive, in that usability, and possibly understanding, become increasingly difficult [Sørumgård93].

TERMSPACE AND VOCABULARY CONTROL

The values that can be used within a facet are members of a set of legal terms known as a termspace [Prieto-Díaz91]. Development and maintenance of a termspace is a complex task, due to the nature of the words that must be used: a word or phrase can never be a perfect representation of the concept for which it stands; the classification is driven by nomenclature rather than meaning [Chen93]. If the termspace is limited to a few terms, the advantage is gained that a user can be very familiar with possible

---

[a] *Faceted schemes are weakest when the facets are not completely orthogonal and it is deemed important to eliminate impossible combinations from the classification scheme. For example, a faceted scheme would not be able to state that classifications such as* (algorithm:Smith's solution of a system of nonlinear equations; language:COBOL) *or* (application:word processing; language:APL) *might be contradictory.*

classifications and therefore decide with a fair degree of confidence which terms may be applicable. However, a small termspace may mean that each term has an undue breadth of meaning so that a diverse group of components may share a single term. As the termspace grows, the problems change: it may be difficult for a user to become familiar with a very large termspace; the terms are increasingly likely to overlap, introducing ambiguity [Prieto-Díaz91]. If the vocabulary used when adding to a termspace is not carefully controlled, additional problems of duplication or ambiguity may be added to the classification system [Prieto-Díaz91]: the terms used may be polysemous, so that one term may be used to classify two unrelated concepts; the terms may be partly or fully synonymous, so that related components may be classified differently. Even with a carefully controlled vocabulary, related problems can occur, in that the growth of the library must at times require additions to the termspace, which in turn may require the complicated task of reclassification of the terms already present [Sørumgård93].

Classification of components in a library is essential to provide a conceptual organization of the library. There are, however, limitations to the methods of classification available. According to a study by Frakes, all classification methods are somewhat weak at imparting understanding of the components, and all are approximately equal at enhancing precision or recall in searches. Frakes recommends implementing multiple methods for the purpose of search accuracy, free-text keyword for cost effectiveness, and enumerated for time effectiveness [Frakes94c]. The classification scheme for the Component Library was implemented partly in response to Frakes's recommendations: the classification uses enumerated termspaces for category facets, keywording for a component's specific purpose, as well as free text querying.

3.2.2 Examples of Classification Systems

A great deal of research has been conducted into software classification systems: the most influential have been the methods used by Prieto-Díaz and Freeman [Prieto-

Díaz87] [Prieto-Díaz91], and Booch [Booch87]; recent methods include the REBOOT system [Sørumgård93], and Structured Abstracts [Poulin95a].

PRIETO-DÍAZ AND FREEMAN

The Prieto-Díaz and Freeman system was the first definitive work on the faceted classification of software components; although the original system is somewhat dated, it still provides the basis for most other systems. The system describes components in terms of the functions performed, environment, how functions are performed (realization), and implementation details; the faceted classification deals with the first two of these, and leaves realization and implementation for the component's description [Prieto-Díaz87].

Table 2 – Prieto-Díaz and Freeman classification system

PROGRAM FUNCTIONALITY:
• Function (action: add, compare, sort, etc.)
• Objects (parameters: arguments, arrays, characters, etc.)
• Medium (locales/data structures: array, file, etc.)
ENVIRONMENT:
• System type (assembler, compiler, DB mgr, etc.)
• Functional area (accounts payable, billing, DB mgr, etc.)
• Setting (advertising, auto repair, catalog sales, etc.)

This system is designed to be used with code fragments of 50-200 lines; the system does not anticipate an object-oriented breakdown or encapsulation of any type, and scaling is not addressed. It seems unlikely that this structure would be easily adapted to a variety of component types.

BOOCH

The Booch system is a limited taxonomy intended to classify Ada components. The hierarchy lends itself to the types of packages it needs to address, but has limited room for expansion [Booch87].

Table 3 – Booch classification system

STRUCTURES
• Atomic structures (bits, bytes, words, integers, etc.)
• Composite types (arrays, records)
• Pointer types

TOOLS • Filters • Pipes • Sorting • (etc.) SUBSYSTEMS

Like the Prieto-Díaz system, Booch's system is limited to relatively small components; Booch's system, however, is an enumerated system, and would be even more difficult to use across a variety of components. It is interesting to note that he makes no real attempt to analyze the "Subsystems" category, which is how most of the components found in the Component Library would be listed.

REBOOT

The REBOOT system[a] is a project run by a group of European commercial interests;[b] its design was based on the Prieto-Díaz and Freeman system, but differs in that it is intended to be used for object-oriented components of diverse size and type [Sørumgård93].

Table 4 – REBOOT classification system

• Abstraction (class name) • Operations (methods) • Dependencies (platform) • Parts (component dependencies – internal) • Collaborators (component dependencies – external)

The REBOOT system seems to be effective in classifying how a component is to be connected to the receiving application, but makes no attempt to classify the purpose for the component.

STRUCTURED ABSTRACTS

The Federal Reuse Repository (see Section 3.3.1) uses a faceted classification method called "structured abstracts." This method focuses on imparting reuse

---

[a] *REuse Based on Object-Oriented Techniques.*

[b] *BULL S.A., CAP Gemini, LGI at IMAG, SEMA Group S.A.E., Siemens A.G., QLABS, Networks, TXT, and SINTEF/NTH.*

information as quickly as possible to the user, by limiting the list of component classifiers to a manageable number. Components are classified by subject in addition to the Structured Abstract (the Component Subjects classifications are used in a hypertext browser; the Structured Abstract is used to generate WAIS queries) [Poulin95a].

Table 5 – Structured Abstracts classification system

STRUCTURED ABSTRACT
• Language
• Domain (accounting, GUI, MIS, etc. – frequently not used)
• Function (add, search, etc.)
• Objects (array, tree, etc.)
• Operating System
• Included elements (composite components)
COMPONENT SUBJECTS
• Bit/String Manipulation Components
• Command Line Components
• Communication Components
• Data Structures Components
• File Services Components
• Graphics Components
• Input/Output Components
• Miscellaneous Utilities
• Numeric and Math Packages
• Operating System Interfaces
• Real-Time Components
• Sorting and Searching Routines
• Synchronization Components

This method of classifying components is intuitive, and is claimed to be particularly effective [Poulin95b], and the Component Library reflects this influence. The primary limitation I see here is that classifying components by purpose through the Structured Abstract seems as if it would be difficult: the *function* facet in the structured abstract would need to have an extremely large termspace if the library contains a variety of components. Poulin states that classifying components by domain is only occasionally useful.

3.2.3 Description and Understanding of Software Components

Classification methods seek to organize components and impart a limited level of understanding of those components; classification cannot wholly describe a (non-trivial) component. Extensive descriptions are necessary to explain the workings of a complex component. There is evidence that formal description methods are not as effective in imparting understanding as prose or narrative methods [Maiden92]. In a library of white-box public-domain components, the demands of description are less than in a commercial library, since an analysis of the code — at no cost to the user — is always an option; in commercial libraries, the description must be thorough enough so that the user has a complete understanding of the use and limitations of the component, since the decision based on the description is not merely whether or not to download the component, but whether to purchase it. Unfortunately, the problem of writing prose descriptions of components involves a paradox of technical writing in general: the people most familiar with the technical issues are frequently not the best equipped to communicate those issues, while the people most fluent in the language are likely to lack detailed technical understanding of the issues involved.

3.2.4 Classification and Description in the Component Library

The classification of components seeks to place the components within a logical structure to ease understanding and retrieval. These concepts are closely related: when a person uses a retrieval system and selects or enters a category, a general level of understanding about the category is communicated; later selection or entry will communicate a more specific level of understanding. When communicating with a retrieval system, a user is going through a classification process closely related to the classification process a developer uses when adding a component to the library: when retrieving a component, the user is classifying the concept of a component, rather than

an actual component [Sørumgård93]. Once an abstract[a] for a component is viewed, it is quickly scanned to classify it still further, either to eliminate it from consideration or identify it as worthy of further analysis [Poulin95a]. Only after that step does a close reading occur.

Some aspects of components are well-suited to a faceted method, such as the programming language or operating system: these have a limited, easily enumerable set of possible values, and are relatively orthogonal to each other and to the components' uses. However, none of the classification systems mentioned seems quite adequate to classify *purpose* of general-purpose software. For components of small size, in certain domains, such as mathematical functions or standard computer science algorithms, a set of facets could be very effective. But moderate- or large-sized components do not naturally lend themselves to classification by an enumerated or faceted method (or would rely on an extremely complex (and therefore of debatable usability) taxonomy of use that does not yet exist). For example, a component used to generate reports based on a SQL database could be classified with equal accuracy as a database utility, a business management tool, or an accounting report — it would probably be appropriate to list it as all three. In reaction to various warnings that a particularly complex classification system has such significant usability problems that it has no benefit over simple systems [Frakes94c] [Poulin95a] [Sørumgård93], it was decided that to classify the purpose of a component, a formal taxonomy would be used to classify the component's *general* purpose, but the *specific* purpose would be classified using a variation of a keyword approach.

The classification system used in the Component Library is a mix of several systems. The component's *purpose* is classified in general by a *General Purpose* facet, which (currently) has a termspace of fourteen categories; a component is classified by one or more of these general categories. The General Purpose categories, shown in Table 6, are loosely adapted from the FRR Component Subjects (see Table 5 for

---

[a] *Since no standard nomenclature for this exists. I will use the term "abstract" to refer to the complete documentation for a component, which includes the component's classification and the prose description.*

comparison), modified to focus more on the needs of business component reuse (determined by scanning catalogs of commercial and shareware VBX and OLE components).

Table 6 – General Purpose categories in the Component Library

- Business and Accounting
- Client Server
- Connectivity, Network, Communications
- Data Structures
- Database
- File Services
- Graphics
- Mathematical Algorithms
- Miscellaneous Utilities
- Multimedia, Audio-Visual
- Operating System, Memory Management
- Software Development
- User Interface
- Word processors, Editors, Electronic Publishing

Classifying more specific purposes is difficult. Several different variations of multiple levels of purpose classifications were designed for the Component Library, but all were found to be impractically complex. The solution implemented is a keywording scheme, but one driven by the General Purpose category for the component: when classifying a component, the developer is presented with a list of keywords that have been used previously in conjunction with that General Purpose category; the developer is encouraged to select from the list, but may enter new keywords, which would be available for selection in the future. The developer would be encouraged to select or add as many terms as possible to extend aliasing of the component's classification, and attempt to lessen the vocabulary problem [Furnas87]. This method is akin to a faceted method, in that this may be seen as a *Specific Purpose* facet, in which the termspace is based on the General Purpose category, and the termspace is only loosely controlled.

The component's *type* is classified with a simple faceted approach: enumerated values are selected for *Type* (source code, OLE component, VBX component, etc.), *Programming Language* (only meaningful if the type is source code), and *Operating System* (only meaningful for some components). Other facets were designed for the

Component Library, such as dependencies, input, and output, but were found to add unnecessary complexity to the classification system, as they are more useful as description rather than classification; these were implemented as structured fields in the component's description.

The component's *description* consists of a detailed prose explanation of the workings of the component, and a series of attribute-value fields that serve to structure the descriptive information. The distinction between the classification facets and the description fields lies in the fact that the classification facets have a controlled (or partially controlled) termspace, and can be used as selections in the browser; the description fields are optional, and can contain any values. A component's description can be seen in Figures 8 and 9.

THE PROCESS OF CLASSIFICATION

The main advantage of free text keywording is the ease of initial classification; its main limitation is the ambiguity of initial classification. A keywording assistance tool has been designed for the Component Library in an attempt to counter the limitations of free text keywording without creating an undue burden for the classifier. Since, as stated above, the processes of initial classification and classification for retrieval are closely related, the tools used should be similar. In the Component Library, initial classification is carried out in a tool based on the browser. The user would enter the component's name and select the general purpose (or purposes), and submit that information (Figure 12); the system would then provide a list of all keywords that occur in conjunction with the component's general purpose throughout the library's collection, and allow entry or selection of all classification information (Figure 13).

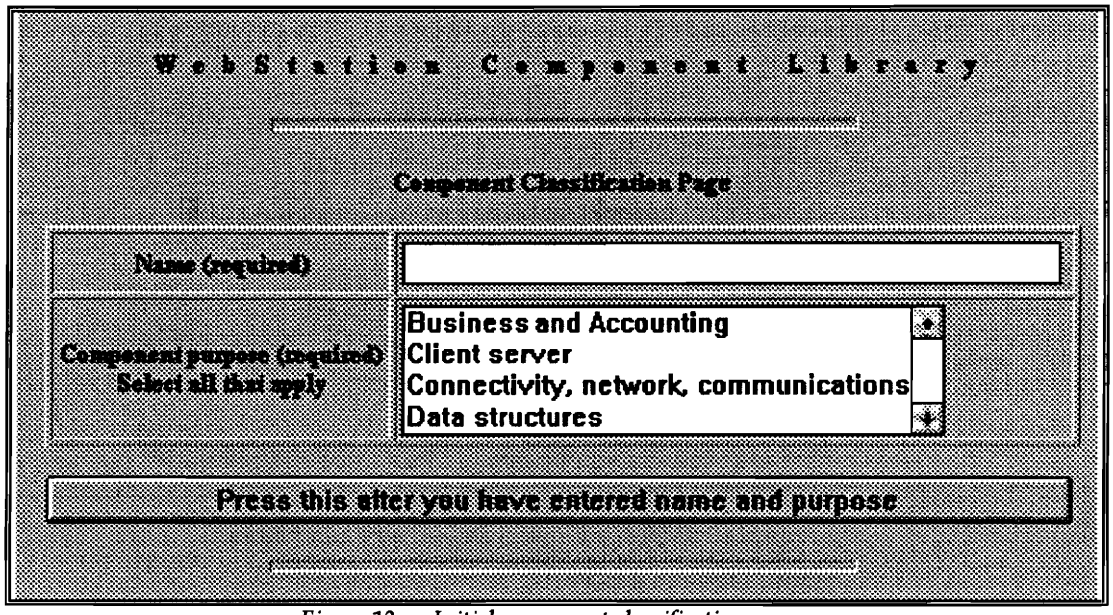

Figure 12 – Initial component classification screen

Name	Accounting Report Tool for SQL
Component purpose	Business and Accounting; Database
Overview	
Keywords (required) Select all that apply	Accounting Charts Data display Dbase
Enter new keywords (optional) separated by semicolons (;)	
Component type (required)	Source code
Programming language (optional)	(no selection)
Operating system (optional)	(no selection)
Description	

Figure 13 – Main classification screen

3.3 Browsing and Querying Methods

There are two primary methods of library interaction: querying and browsing. The purposes of these methods are somewhat different: in querying (searching), the goal is to determine if a particular component, or type of component, is present; in browsing, the goal is to determine more generally what is present [Henninger94] [Poulin95a] [Sørumgård93]. This distinction is not absolute: a user will frequently use a querying method to gain access to a relatively specific subject area, then browse within that area either to determine what is present, or to use the retrieved components as a potential source of inspiration. Several issues determine whether library interaction is best supported by querying or browsing [Henninger94] [Poulin95a]:

- small collections are more amenable to browsing;

- some types of development are demanding of particular solutions, while others are reactive and tend to build off of what is present;
- at early stages of development, user needs may be too inchoate to allow accurate querying;
- classification systems that present the user with long lists of categories discourage potential browsers;
- novice users are daunted by querying, while expert users prefer querying.

A great deal of research has been conducted on methods of querying, both for libraries in general and reuse collections in particular (e.g. [Chen93] [Fugini93] [Karlsson92a]); a variety of querying tools are generally available (such as WAIS). Browsing, however, has received much less attention. For this reason, it seems appropriate to focus work on browsing mechanisms and to incorporate an existing querying scheme into the library interface; querying capabilities in the Component Library have been implemented by using a version of WAIS.

The method of browsing depends on the size of the library. In a library of a limited size, the components themselves may be browsed; the user scans through lists of the components and selects those of interest. In a larger system, however, the components cannot be browsed directly, or a form of "option paralysis"[a] will set in; the *classification* of the components must be browsed. A strictly hierarchical classification system simplifies browsing in this way, but, as shown above, is not an accurate method of representing general-purpose components. Therefore, the classification system must be developed in conjunction with a browsing method so that the user is never presented with an unmanageable list of possible options. For many people, the preferred retrieval method is to narrow the search until a reasonable number of components is left, then to scan directly through that list.

---

[a] *The inability to make a decision when confronted with an overwhelming number of options.*

3.3.1 Examples of Browsing and Querying Interfaces

CODEFINDER

The CodeFinder interface uses iterative refinement as a method of query formulation, in the belief that ineffective query construction is a significant limitation of retrieval systems. When a query is constructed, the system returns not only a ranked list of possible components, but a ranked list of related query terms; by selecting the suggested terms, a complete and accurate query can be generated [Henninger94].

This method of query formulation seems particularly powerful. The method of browsing based on iterative refinement was important in the design of the browser for the Component Library. Codefinder uses a very complex user interface, which would presumably detract somewhat from its usability; the Component Library attempts to allow iterative refinement in a more straightforward manner.

FEDERAL REUSE REPOSITORY[a]

The information in the Federal Reuse Repository can be accessed either through a 2-level hierarchy, or through a query mechanism based on "structured abstracts" (see Section 3.2.2) The hierarchical browser is a simple hypertext implementation. The query mechanism is based on the belief that reuse information is best communicated through a narrative form, and is structured in this way: "I need a <select from list> language component for <select from list> (domain) that performs the <select from list> function on <select from list> objects, that runs on the <select from list> operating system and that also <query (text entry)>." [Poulin95a]

The interface Poulin describes seems quite useful, but since the "function" list is necessarily quite long, retrieving components by use is likely to be inaccurate. A limitation of current Web browsers is that menus such as these are not truly interactive: a user must make all choices, then submit the form. Since some of the lists can be quite long (and therefore frustrating to select from), it would be useful if each selection

---

[a] *FRR is a Department of Defense funded project begin implemented at LORAL (formerly IBM Federal Systems); since the contents are classified, the library is not publicly accessible.*

limited the possible options on the other lists to the possible combinations of attributes present in the library. Unfortunately, such interactive selection in a Web browser is only possible through the use of lists of hypertext links, which would greatly extend the length of the Web pages, to the detriment of usability. The interface in the Component Library attempts to balance these conflicting demands, in that it uses pull-down menus to conserve screen space, but limits the content of the lists of terms in reaction to previous selections.

NIST GUIDE TO AVAILABLE MATHEMATICAL SOFTWARE (GAMS)

GAMS is the current means of access for the venerable netlib catalog of mathematical algorithms. Since the classification system used is a strict enumerated hierarchy, the interface lends itself to a hypertext browser, which is provided in addition to a querying system [GAMS94]. If the contents of the categories in the hierarchy were more apparent, this approach would be more usable: for example, determining that sorting algorithms can be found in section "N: Data Handling" is not a certain process.

3.3.2 Browsing and Querying in the Component Library

The method of browsing the Component Library is in part derived from features of the Federal Reuse Repository and CodeFinder, as described above. The *Programming Language* and *Operating System* facets were borrowed from the Federal Reuse Repository, as well as some of the categories, and the idea of using HTML menus to allow selection of terms for all facets in a single screen. The concept of using interactive refinement is based on CodeFinder.

The browsing method implemented in the Component Library is based on the method of classification used. Each of five facets (*Type, Language, Operating System, General Purpose,* and *Specific Purpose* — see Figure 4) can be used as selections in the browser. When selections are submitted, the termspace for the other facets is limited to the terms that appear in conjunction with those selections. For example, a user interested only in OLE components selects OLE as the component *Type* and presses the

submission button; the system regenerates the form with new values, so that the only possible selections for *Operating System* might be Windows 3.1, Windows NT 3.5, and Windows 95, and the only possible selections in the *General Purpose* facet might be Client Server, Database, and Miscellaneous Utilities. Note that selections limit the relevant components to only those documents for which *all* selections are true — the browser operates as a boolean AND. Using the browser as a method of limiting the selections in such an absolute way is based on the methods of browsing used by some people [Poulin95b], but evaluation of the prototype system has shown that a fuzzy AND, with ranked feedback, might increase usability (see Section 3.5).

When the list of components limited by current selections contains more than forty items, the list is not displayed on the current page, but is available for display (see Figure 1). This decision was made to limit display time, in the belief that long unranked lists do not tend to be useful; the choice of the number forty was based on the default length of WAIS feedback lists. Ranked lists of components, such as are returned by queries, display the forty most relevant components.

QUERY WEIGHTING

The querying mechanism uses freeWAIS to search the keyword and description files. Separate searches of the keywords and descriptions are merged together so that a keyword match can be weighted more than a description match. The method of weighting in freeWAIS is not quite appropriate for these purposes and had to be adjusted. freeWAIS calculates the weights of documents for ranking based on the frequency of terms in a given document, the inverse frequency of terms in the entire collection, the position of terms in the document, and the length of the document [freeWAIS]. When the length of documents varies substantially, as is true with the Component Library descriptions, freeWAIS overcompensates for the length of the document, so that brief descriptions are always ranked substantially higher than lengthy descriptions. For example, a search for the term "database" across a set of keyword files would return a perfect match if "database" was the only term in the list, but a 40% match if "database" was the first term out of ten terms. To balance this bias against long files, the Component Library reweights the results returned by freeWAIS.

In short, the Component Library places more emphasis than freeWAIS on attaining any match at all. The calculation to adjust and merge the freeWAIS searches is implemented in this way:

- since matches in keyword files should always be ranked highly, a nonzero freeWAIS score for the keyword search is averaged with a 100% score;
- to emphasize matches in description files, a nonzero freeWAIS score for the description search is averaged with a 100% score, with the description score receiving a triple weight;
- the modified keyword score is treated as 70% of the final score, and the modified description score is treated as 30% of the final score.

Therefore, if K is the score for a freeWAIS search of a component's keyword file and D is the corresponding description file score, then the score used in the Component Library is

$$\left(0.7*\left(\frac{K}{2}+\frac{1}{2}\right)\right)+\left(0.3*\left(\frac{3D}{4}+\frac{1}{4}\right)\right)$$

For example, if on a search for the term "database" component A had a 100% match for its keyword file and a 70% match for its description file, and component B had a 40% match for its keyword file and a 10% match for its description file, then the weight for component A is

$$\left(0.7*\left(\frac{1}{2}+\frac{1}{2}\right)\right)+\left(0.3*\left(\frac{3(0.7)}{4}+\frac{1}{4}\right)\right)=0.7+0.2325=93.25\%$$

The weight for component B is

$$\left(0.7*\left(\frac{0.4}{2}+\frac{1}{2}\right)\right)+\left(0.3*\left(\frac{3(0.1)}{4}+\frac{1}{4}\right)\right)=0.49+0.0975=58.75\%$$

A variety of calculations were tested before this algorithm was implemented. Weighting the keyword and description searches equally returned inappropriate values, since a keyword match is almost certainly of value, while description matches may be of greater or lesser value. Basing the scores on various functions of file length were attempted, but the primary drawback of the freeWAIS weights is that a minimum

match is not weighted strongly enough; simply increasing the weights as shown above was found to return logical values.[a]

Querying can be used either separately or in conjunction with browsing as a selection method, as is shown in the following scenario:

SCENARIO # 3 (COMBINATION OF BROWSING AND QUERYING)

- David wants to know what components in the library interact with DBase.
- He enters "DBase" in the search field; the system redisplays the browser, showing that the only applicable Types are VBX and OLE components, and the only applicable General Purposes are "Business, accounting, and spreadsheet" and Database. The forty most relevant components are displayed at the bottom of the page.
- He selects OLE as the Type, and Windows 95 as the Operating System; the system redisplays, showing a list of the OLE / Windows 95 components, ranked by the frequency of the term DBase.

If David had selected the values first, then conducted the search, he would have received the same results.

3.4 Scaling

The size of a library, measured by number of components, has a substantial impact on how the library is used. Very large libraries (either digital or physical) tend to be daunting to users, as the complexity of the interface tends to increase with the size of the library. User behavior also is affected: small libraries can be browsed in their entirety; with large libraries, users are more dependent on the classification and retrieval systems.

---

[a] *The purpose in the reweighting was to counter the values returned by freeWAIS, which were patently inappropriate; no more scientific method of analysis was used than judging what seemed to be intuitively reasonable scores for specific components.*

Preliminary testing of the Component Library has used a database that is composed of some abstracts written as examples of appropriate component description, but also of "synthetic" abstracts, generated so that each node on the prototype taxonomy has at least one component; the use of this method allows testing on a database of approximately 500 components, which could not practically be generated by hand. The results from preliminary evaluation of alternative classification and browsing methods assisted in developing the current design. In particular, preliminary evaluation showed that a three level classification method for the component's purpose (category, subcategory, and keyword) was unwieldy and introduced ambiguity, both during classification and browsing; hypertext lists of longer than one or two screens of classification terms (as would be necessary with hundreds of components) were found to be too daunting to be useful.

3.5 Design History

Since the Component Library may be used as a basis for future work, it is appropriate to include an outline of its design history, including the rationale for some decisions made during design or implementation.

THE WORLD-WIDE WEB

The Component Library was intended from the start to be used over the World-Wide Web: the success of such Web browsers as Mosaic or Netscape provided the inspiration for the system in that it is the first standard, widely used client interface that can be used in conjunction with a variety of server applications. Implementation of the Component Library in any other way would severely limit the accessibility of the library.

The use of HTML restricts the display of information in some ways, but is only a drawback in that pull-down menus cannot be used interactively, but must be submitted as part of a form.

CLASSIFICATION

The initial desire was to find an existing classification system that could be used without modification. After examining all of the classification systems I could find (the most notable are shown in Section 3.2.2), I decided that none of the existing systems were appropriate for use here. I next attempted to derive a classification system in which the component's type (language, operating system, external dependencies, etc.) would be structured by an enumerated system, and the component's purpose would be structured by another enumerated system. Recommendations made by Frakes and Prieto-Díaz [Frakes94a] [Prieto-Díaz91] made me doubt the wisdom of this approach.

The first classification system that was actually implemented in the Component Library was a faceted system that used a large number of facets, not just including the current type, language, etc., but also those attributes that are now included in the structured description, such as input, output, compression method, and version number. It soon became apparent that most of these facets were not appropriate for classification, but as attributes for description, as they added needless complexity to the classification system.

The classification of purpose is the most difficult part of a component's classification. I spent a great deal of time attempting to develop a system that used three tiers of purpose facets. In that system, each general purpose (equivalent to the current general purpose) would include a specified termspace for a "purpose subcategory" facet, the value for each of which would include a specified termspace for a specific purpose facet. That system was conceptually complex, in that the distinctions between the levels of purpose were often not apparent. That system also was not truly a faceted system, but a variety of both a faceted and enumerated hierarchical system, but without the advantages of either. The current purpose classification method specifies a limited termspace for general purpose, but uses keywording for specific purpose. This allows the advantage of two systems: the flexibility of classification and retrieval based on keywording is maintained, but the specific purpose can be used as a facet orthogonal to general purpose for classification and retrieval based on a faceted system.

BROWSING

The browser in the Component Library is simply a reflection of the classification system. The browser was implemented as a method of rapidly limiting the search criteria in a way I personally use, and which Poulin cites as effective [Poulin95b]. For this reason, the browser functions as a boolean AND. This decision, however, may be seen as limiting: it reflects the browsing behavior of only some possible users, and does not allow the ranking of a returned list of components. Future enhancements to the system may include the implementation of a fuzzy AND for processing of browser selections.

The browser was initially implemented as a series of lists of hypertext links, which allowed the system to react immediately to every user selection. However, more than one or two facets could never be displayed on the screen at one time, and some long termspaces would stretch over several screens in the Web browser. This was clearly unacceptable: the use of HTML menus allowed conservation of screen space, but at the expense of system responsiveness.

The feedback from the browser was originally to be ranked (even though the browser operates as a boolean AND) based on where in a component's list a given term was matched. For example, if a component had keywords a,b, c, and d in its keyword list, in that order, then a browser selection of term a would return a perfect match, while a browser selection of term d would return a partial match. This method of ranking proved to be far too confusing to be usable.

ITERATIVE REFINEMENT

The idea of using iterative refinement of selections or queries as a browsing method was part of the system from its inception, and predated my reading of Henninger's implementation of an iterative refinement method in CodeFinder [Henninger94].

QUERYING

The decision to use freeWAIS to process queries was based on its availability. Incorporating freeWAIS queries into the system was relatively straightforward, and

proved to be good in terms of response time. The method by which freeWAIS weights its responses was frustrating, however, as shown in Section 3.2.2: querying would be enhanced if freeWAIS allowed weighting methods to be selected or parameterized.

4 FUTURE WORK

The Component Library and the analysis of issues in this report are intended to be used as the basis for future work, in which the system will be elaborated and refined in response to the work presented here, and developed as an operational commercial concern. This future work will be conducted in two phases: system development and collection building. In the system development phase, this hypothetical future system will be developed, creating a market-ready library system, with state-of-the-art library browsing and searching facilities, a classification system that facilitates both retrieval and understanding, secure credit transactions over the World-Wide Web and reliable policies on quality assurance, liability, copyright, and marketing. After the system development phase is complete, the library could be operated commercially at a limited level, and would be able to grow to some extent by "bootstrapping" – by using revenue from the first few components in the library as capital in expanding the library. For the library to be successful, however, it must have enough components to interest prospective users. To generate this critical mass of components, a concerted effort must be made to acquire and develop components; this would be the purpose of collection building, the second phase of future work. During the collection building phase, a substantial number of components must be acquired, tested for reliability, classified, and documented; the collection building phase, of course, will never be complete, as the acquisition of new components must continue as long as the library exists.

Some of the issues analyzed in the design of the Component Library will need to be reevaluated for the library to be commercially implemented. Other open issues need to be addressed by future work. A summary of these issues is presented here.

CLASSIFICATION AND DESCRIPTION

The success of a classification and description scheme can be measured by its effectiveness in four areas:

- ease of accurate classification,
- usability and accuracy of retrieval tools based on the scheme,
- user understanding of the component, and
- scaling as the library grows.

Future research is needed to assess the usability of the classification and description methods implemented in the Component Library: a formative evaluation and redesign cycle will be conducted, focusing on these four areas. It may be seen as useful to conduct an evaluation of understanding based on classification and description methods, although research indicates that no classification method has been successful at imparting understanding [Frakes94c].

QUERYING

The querying mechanism as implemented does not account for synonymous terms. freeWAIS does allow searches based on a thesaurus (which must be constructed); to increase the success of queries in the Component Library, a thesaurus should be implemented.

After taking synonymous terms into consideration, the Component Library's querying mechanism should be fully functional. Later evaluations should be conducted, however, to indicate whether added functionality, such as relevance feedback or boolean queries, should be added to the system.

SCALING

The current Component Library is designed to be usable for libraries including up to several hundred components. Future extensions to this library should be prepared to manage a library of several thousand components. Testing for scaling problems might be conducted on a large library of automatically generated component descriptions, similar to those generated for testing the classification scheme and user

interface for the current system; the classification method and the browsing and querying interface may need to be reevaluated in light of such testing.

ACQUISITION AND INCENTIVES FOR REUSE

For libraries of reusable software components to thrive, there must be economic incentives to provide components for the library, to maintain the library, and to use the components. The standard incentive for the population and maintenance of generally accessible reuse libraries, ever since McIlroy, has been government funding: "The first source of support that comes to mind is governmental, perhaps channeled through semi-independent research organizations." [McIlroy69] A number of corporations have developed in-house libraries, but, according to Jeffrey Poulin's dystopian view, most of these libraries pass through a three-stage development cycle:

Phase 1 : very few parts

Phase 2 : many parts of low or poor quality

Phase 3 : many parts of little or no use [Poulin95c]

Japanese software corporations have had much-touted success with reuse: the reasons for this lie in the fact that much of the software being produced in these particular companies is highly redundant custom software, and that the companies pay their developers per line of code added to or removed from the libraries. For general purposes, though, there is doubt about how frequently components of any size are likely to be used, particularly in that studies have indicated that components must be reused more than 13 times to be cost effective [Favaro91].

Since the Component Library will be consignment-based, there is some expectation that the library can be built by "bootstrapping" — by using revenue from the first few components in the library as capital in expanding the library. There are several difficulties with this approach, however. The process of developing a library in this way is at best slow, and stands a good chance of failure. If the library is developed slowly, an opportunity will be lost in that competition will be allowed to develop; a great part of the success of such a library is dependent on being recognized as a

standard. Therefore, part of the future research agenda is an analysis of how library growth can be maximized, particularly in the early stages.

RELIABILITY

The reputation, and therefore the success, of a component library is built upon faith in the reliability of components. Unfortunately, the cost of thorough testing of components is extremely high; if the library is to be run on a limited margin, any attempts to guarantee reliability will be unaffordable. In the early stages, quality assurance must (unfortunately) be *caveat emptor*. The problems with this approach can be lessened, however, in a few ways. The library would have a money-back guarantee to a customer who can show that a purchased component did not work properly, or was described improperly; it would be incumbent on the developer to correct the problems. Developers would be encouraged to include testing procedures and test data along with the component. For white-box components, source code can be required to be well documented. The description of the components will include the number of versions sold, in the belief that frequently-sold components are likely to be more trustworthy than little-used ones; likewise, the first few copies of any component would be sold at a discount. At some point in the distant future, in-house testing or an independent testing firm may be used, but if the library will have to remain a low-overhead operation to succeed, that seems unlikely.

Reuse, particularly of black-box components, presents more reliability concerns even than regular software development. Reuse requires placing trust in a stranger. If the developer who provided the component is well-known, a certain amount of doubt is assuaged; in a library such as this one, developers must remain anonymous, and therefore unknown.[a] Providing developer names without any access information may be practical, but improving Internet and Web search tools will probably be able to provide access information for most people with Internet accounts. Developer names

---

[a] *Otherwise the library would simply be acting as advertising for the developer; users would get the name from the library, contact the developer directly, and buy the component without paying the (expected) 10-15% overhead for the library.*

could be provided for developers who agree not to independently distribute their components.

If a great deal of the code in the library is developed by hobbyists, the reliability of that code may be problematic. Reusing black box components is extremely doubtful: since the source code is not available, there is no way to verify that the component will not have unintended or malicious effects; similarly, if a component works properly for its intended purpose, but is reused in another way, it may be unreliable. Solutions to these concerns are not apparent. Quality assurance is likely to be a major complication for the operation of the Component Library.

PAYMENT SECURITY

Information traveling as packets over the Internet is not secure; if a site is known to receive a great deal of credit card number transactions, it is not unlikely that the majority of packets destined for that site will be tapped. Secure credit card transactions over the World-Wide Web can be managed by using a Netscape Commerce Server, which requires a Netscape browser; other Web servers using the public domain Secure HTTP protocol[a] will be available soon. Although other solutions are available, such as PGP encryption, electronic cash, and virtual banks, all of these detract from the convenience that is the raison d'être for the Web [Wiggins95].

The solutions available without resorting to such security methods are straightforward: some firms allow unsecured credit card transactions over the Web, stating, somewhat disingenuously, that such transactions are no less safe than leaving a credit card slip on a restaurant table, or repeating a credit card number over a telephone; customers could be required to register by telephone, fax, or mail, after which passwords could be used in lieu of credit card numbers to authorize transactions.

COPYRIGHT AND LICENSING

The Component Library will need to set policies on copyright and licensing of the components it sells. Some of the issues that need to be addressed include whether

---

[a] *The Secure HTTP protocol was defined by Teresa Systems (a company formed by NCSA, Enterprise Integration Technologies (EIT) & RSA Data Security) in 1994.*

the copyright for the components is retained by the programmer, or somehow shared by both the programmer and the library. Some restrictions should be placed on the use of components: components should be licensed so that they cannot be resold in component form, or reverse engineered; the license should provide that the component can be used to build executables that can be distributed without restriction, but that the component cannot be distributed in any other form.

5 CONCLUSIONS

No known services are directly competitive with this system. Several reuse libraries are being developed that focus on public-domain components of a more academic bent, including ASSET [ASSET], GAMS [GAMS94], and MORE [Eichmann94c]; FRR [Poulin94] has limited access for security reasons. None of these libraries is likely ever to focus on business-level components, and so are unlikely ever to be direct competition. Business-level components are available as shareware over the Internet or bulletin boards, or commercially through (paper) catalogs; the former tend to be limited in size and value, and the latter suffer from limited selection, and limited distribution. The library will be successful in moving from research to the market if it acquires a critical mass of components; the initial components will probably be acquired through agreements with those developers who currently market their components through catalogs. Since supplying components to the library would not cost the developers anything and would have the possibility of being fairly profitable, much of the response in the early stages is likely to be from hobbyists, and although there are drawbacks to focusing on that audience exclusively, it should not be ignored.

Whether or not this particular library is successful, the research conducted in this direction will be valuable in furthering the process of software development in general, as future libraries will build upon this work; there is good reason to believe that such commercial libraries will have a substantial impact on the process of software development.

REFERENCES

[ASSET] (anonymous); "Asset Source for Software Engineering Technology"; http://source.asset.com/.

[Belanger94] Belanger, David, and Balachander Krishnamurthy; "Practical Software Reuse: An Interim Report"; *Proceedings, Third International Conference on Software Reuse: Advances in Software Reusability*; Rio de Janiero, Brazil; November 1-4, 1994; p. 53-6.

[Biggerstaff89] Biggerstaff, Ted J., and Charles Richter; "Reusability Framework, Assessment, and Directions"; *Software Reusability, Vol. 1: Concepts and Models* (ed. Biggerstaff and Perlis); ACM Press; 1989, p. 1-18.

[Boisvert 94] Boisvert, R.F.; "A Web Gateway to a Virtual Mathematical Software Repository"; *Electronic Proceedings of the Second International World-Wide Web Conference*; Chicago, Illinois; October 17-20, 1994.

[Booch87] Booch, Grady; *Software Components with Ada; Structures, Tools, and Subsystems*; Bejamin/Cummings; 1987.

[Caldiera91] Caldiera, Gianluigi, and Victor R. Basili; "Identifying and Qualifying Reusable Software Components"; *IEEE Computer*; February 1991; p. 61-70.

[CARDS] (anonymous); "A Comprehensive Approach to Reusable Defense Software (CARDS)"; http://dealer.cards.com/.

[Chen93] Chen, Patrick Shicheng, Rolf Hennicker, and Matthias Jarke; "On the Retrieval of Reusable Software Components"; *Proceedings: Advances in Software Reuse; Selected Papers from the Second International Workshop on Software Reusability*; Lucca, Italy; March 24-26, 1993; p.99-108.

[CNIDR94] Gamiel, Kevin; "Why We Changed the Name freeWAIS to ZDist"; March 16, 1994; http://cnidr.org/talks/whyzdist.html.

[CNIDR95] (anonymous); "CNIDR Software Development Group"; http://vinca.cnidr.org/software/software.html.

[Coombs87] Cooms, James, Allen Renear, and Steven DeRose; "Markup Systems and the Future of Scholarly Text Processing"; *Communications of the ACM*; Volume 30, Number 11; November 1987; p. 933-947.

[DSRS] (anonymous); "The Defense Software Repository System (The DSRS)"; http://ssed1.ims.disa.mil/srp/dsrspage.html.

[Eichmann94a] Eichmann, D., T. McGregor and D. Danley, "Integrating Structured Databases Into the Web: The MORE System," *First International Conference on the World-*

Wide Web; Geneva, Switzerland; May 25-27, 1994; http://rbse.jsc.nasa.gov/eichmann/www94/MORE/MORE.html.

[Eichmann94b] Eichmann, David; "The Repository Based Software Engineering Program"; *Proceedings of the Fifth Systems Reengineering Technology Workshop*; Monterey, CA; Feb. 7-9, 1995; http://rbse.jsc.nasa.gov/eichmann/rbse_desc.html.

[Eichmann94c] Eichmann, David; "MORE: A Case Study in Reengineering a Database Application for the Web"; *Proceedings of the Second International World-Wide Web Conference: Mosaic and the Web*; Chicago, IL; October 18-20, 1994; http://rbse.jsc.nasa.gov/eichmann/www-f94/reeng.html.

[Favaro91] Favaro, J.; "What Price Reusability? A Case Study"; *Ada Letters*; March 1991; p. 115-124 (as cited in [Frakes94b]).

[Frakes92] Frakes, William; "Introduction to Information Storage and Retrieval Systems"; *Information Retrieval: Data Structures and Algorithms* (ed. Frakes, Baeza-Yates); Prentice Hall; 1992; p. 1-12.

[Frakes94a] Frakes, William B., and Thomas P. Pole; "An Empirical Study of Representation Methods for Reusable Software Components"; *IEEE Transactions on Software Engineering*; Volume 20, Number 8; August 1994; p. 617-630.

[Frakes94b] Frakes, William B., Sadahiro Isoda; "Success Factors of Systematic Reuse"; *IEEE Software*; September 1994; p. 14-19.

[Frakes94c] Frakes, William; "Systematic Software Reuse: a Paradigm Shift"; *Proceedings, Third International Conference on Software Reuse: Advances in Software Reusability*; November 1-4, 1994; p. 2-3.

[freeWAIS] (anonymous); freeWAIS version 0.3 documentation.

[Fugini93] Fugini, M.G., and S. Faustle; "Retrieval of Reusable Components in a Development Information System"; *Proceedings: Advances in Software Reuse; Selected Papers from the Second International Workshop on Software Reusability*; March 24-26, 1993; p. 89-98.

[Furnas87] Furnas, G. W., T. K. Landauer, et al.; "The Vocabulary Problem in Human-System Communication"; *Communications of the ACM*; Volume 30, Number 11; November 1987; p. 964-971.

[GAMS94] NIST Guide to Available Mathematical Software; http://gams.nist.gov/.

[Harman92a] Harman, Donna, Edward Fox, et al.; "Inverted Files"; Information Retrieval: Data Structures *and Algorithms* (ed. Frakes, Baeza-Yates); Prentice Hall; 1992; p.28-43.

[Harman92b] Harman, Donna; "Ranking Algorithms"; *Information Retrieval: Data Structures and Algorithms* (ed. Frakes, Baeza-Yates); Prentice Hall; 1992; p.363-392.

[Henninger94] Henninger, Scott; "Using Iterative Refinement to Find Reusable Software"; IEEE *Software*; September 1994; p. 48-59.

[Karlsson92a] Karlsson, Even-Andre, et al; "Weighted Term Spaces for Relaxed Search"; *CIKM'92*; Baltimore, MD; 5-8 November 1992.

[Karlsson92b] Karlsson, Even-Andre, et al; "Classification of Object-Oriented Components for Reuse"; *Proceedings of Technology of Object-Oriented Languages and Systems: TOOLS Europe '92*; 1992.

[Krueger92] Krueger, Charles W.; "Software Reuse"; *ACM Computing Surveys*; Volume 24, Number 2; June 1992.

[Maiden92] Maiden and Sutcliffe; "Analogously based reusability"; *Behavior and Information Technology*; March/April 1992; p. 79-98 (as cited in [Poulin95]).

[McIlroy69] McIlroy, M.; "Mass-Produced Software Components"; *Software Engineering: Concepts and Techniques* (ed. Naur, Randell, Buxton); 1976; p. 88-98 (reprint of *Proceedings of the NATO Conference on Software Engineering*; 1969).

[Morgan95] Morgan, Lisa; "Cashing In: The Rush is on to Make Net Commerce Happen"; *Internet World*; February 1995; p. 48-51.

[Naur68] Naur, Peter, and Brian Randall, eds.; *Software Engineering; Report on a Conference by the NATO Science Committee*; 1968 (as cited in [Krueger92]).

[Neighbors94] Neighbors, James M.; "An Assessment of Reuse Technology after Ten Years"; *Proceedings, Third International Conference on Software Reuse: Advances in Software Reusability*; November 1-4, 1994; p. 6-1

[Netscape] Netscape Corporation, "Extensions to HTML"; http://www2.netscape.com/assist/net_sites/html_extensions.html.

[Ning94] Ning, Jim Q., Kanth Miriyala, and W. Kozaczynski; "An Architecture-driven, Business-specific, and Component-based Approach to Software Engineering"; *Proceedings, Third International Conference on Software Reuse: Advances in Software Reusability*; November 1-4, 1994; p. 84-9

[Petzold92] Petzold, Charles; *Programming Windows 3.1, Third Edition*; Microsoft Press; 1992.

[Poulin93] Poulin, Jeffrey S, and Kathryn P. Yglesias; "Experiences with a Faceted Classification Scheme in a Large Reusable Software Library (RSL)"; *Seventeenth Annual International Computer Software and Applications Conference (COMPSAC'93)*; Phoenix, AZ, 3-5 November 1993, p 90-99.

[Poulin94] Poulin, Jeffrey S. and Keith J. Werkman; "Software Reuse Libraries with Mosaic"; *Proceedings of the Second World-Wide Web Conference '94: Mosaic and the Web*; http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/werkman/www94.html.

[Poulin95a] Poulin, Jeffrey S. and Keith J. Werkman; "Melding Structured Abstracts and the World-Wide Web for Retrieval of Reusable Components"; to appear, *ACM SIGSOFT Symposium on Software Reusability (SSR '95)*, Seattle, Washington, April 28-30, 1995.

[Poulin95b] Poulin, Jeffrey S; Email correspondence; Feb-Apr 1995.

[Poulin95c] Poulin, Jeffrey S; "Populating Software Repositories: Incentives and Domain-Specific Software"; to appear, *Journal of Systems and Software*; Spring 1995.

[Press94] Press, Larry; "Commercialization of the Internet"; *Communications of the ACM*; Volume 37, Number 11; November 1994; p. 17-21.

[Prieto-Díaz87] Prieto-Díaz, Rubén, and Peter Freeman; "Classifying Software for Reusability"; *IEEE Software*; January 1987; p. 6-16.

[Prieto-Díaz91] Prieto-Díaz, Rubén; "Implementing Faceted Classification for Software Reuse"; *Communications of the ACM*; Volume 34, Number 5; May 1991; p. 88-97.

[Ranghanathan57] Ranghanathan, S.R.; *Prolegomena to Library Classification*; 1957 (as cited in [Sørumgård93]).

[Rao95] Rao, Ramana, Jan O. Pedersen, et al.; "Rich Interaction in the Digital Library"; Communications of the ACM; Volume 38, Number 4; April 1995; p. 29-39.

[SBIR95] *Small Business Innovation Research (SBIR) Program Solicatiation*; National Science Foundation; NSF95-59; Closing Date: June 12, 1995.

[Sørumgård93] Sørumgård, Lars Sivert, Guttorm Sindre, and Frode Stokke; "Experiences from Application of a Faceted Classification Scheme"; *Proceedings: Advances in Software Reuse; Selected Papers from the Second International Workshop on Software Reusability*; March 24-26, 1993; p. 116-124.

[Sørumgård95] Sørumgård, Sivert; Email correspondence; April 1995.

[Stroustrup91] Stroustrup, Bjarne; The C++ Programming Language (Second Edition); 1991.

[Udell94] Udell, Jon; "Component"; *Byte*; May 1994; p. 46-56.

[Wiggins95] Wiggins, Richard W.; "Business Browser: A Tool to Make Web Commerce Secure"; *Internet World*; February 1995; p.52-55.