Classifying Software Components Using Design Characteristics*

Chris Clifton† Wen-Syan Li‡
Northwestern University/EECS
Evanston, Illinois, USA

Abstract

Classifying software modules in a component library is a major problem in software reuse. Indexing criteria must adequately reflect the semantics of the components. This must be done without undue effort in either classifying the software, or developing "queries" to find candidates for reuse. We present an architecture for automatically classifying and querying software based on design information. We present a method for determining if indexing criteria are effective, and show results using a set of criteria automatically extracted from an existing collection of programs.

1. Introduction

Software reuse was first introduced at the 1968 NATO Software Engineering Conference. However, software reuse has failed to become a standard practice in software engineering. Most software is custom-built, rather than being assembled from existing components, even though standard components' manufacture and reuse is common practice in other engineering disciplines (e.g., computer hardware uses few custom chips). In order to reuse available software components, these building blocks must be cataloged for easy reference, standardized for easy integration, and validated to reduce development burdens. Software component reuse would allow large software projects to be accomplished with lower development costs.

There are two very different software reuse scenarios:

- Software development as a process of combining modules from a component library built specifically for reuse.
- *Software salvaging*[14], the *re*use of modules written for some specific initial use.

Work based on the first approach has concentrated on building domain-specific software development environments; the component libraries are built with a particular application area in mind. This is more likely to achieve the goal of software development as a process of assembling components than the second approach, however we believe there will still be a need for software outside of the domains of the available component libraries. These applications will require writing custom software (as with traditional software development methods), however such applications can use legacy software modules in the new design. Even if such legacy components cannot be used directly, re-engineering them for the new application may be easier than writing new modules from scratch. Thus our "component library" is created from legacy software.

For software salvaging to succeed, the cost to reuse legacy software must be less than that required to build from scratch. One possible problem is that a development environment supporting salvaging could *increase* the cost for custom development. Our solution is to view software salvaging as a low-cost option to an existing development process. Increased costs associated with a salvaging environment can be divided into two areas: The cost of making modules available for reuse, and that of finding and using existing code in a new design. We will first discuss the problem of making existing code available for reuse.

Since most programmers are charged with making sure that the modules they create are suitable for their *initial use*, they have little incentive to expend effort in making their modules available for *reuse*. This is not to say that they will not write reusable code. Many other considerations, such as maintainability and portability of the initial application, will likely lead to modules that are appropriate for reuse. However, we cannot expect the author of a module to expend effort in "advertising" it. Therefore, we impose the constraint that adding software to a reuse library be automatic.

Simply making source code widely available satisfies this constraint, however requiring a programmer to manually hunt for reuse candidates is unreasonable (particularly if no such candidates exist for a given design). This gives us a second constraint: Searching for reuse candidates must be a no (or low) cost addition to the custom development process.

This paper focuses on development of software module indexes using information that meet the following criteria:

- Indexing information can be automatically extracted from software.
- *Keys* for searching the index can be automatically determined from the design process. This can be used in conjunction with other design-level retrieval methods[6].

This enables software developers to treat reuse of legacy modules as a "value-added" feature applied to their existing software development process, rather than an expensive effort of dubious value.

* This material is based upon work supported by the National Science Foundation under Grant No. CCR-9210704.

† Author's current address is The MITRE Corporation, Bedford, MA 01730-1420, clifton@eecs.nwu.edu.

‡ Author's current address is CIMIC, Rutgers University, 180 University Ave, Newark, NJ 07102, wsli@andromeda.rutgers.edu.

Our idea is to automatically extract certain *discriminators* from software modules. These discriminators are information (such as type signatures) that are somehow dependent on the semantics of the module. We can then query the modules available for reuse by providing a set of discriminators, and searching for modules with a similar set. These discriminators must be available from both the legacy modules and from a new design. Automatic extraction of the discriminators from legacy modules allows us to meet the first constraint; that adding modules to the software library be extremely low effort. Our approach to meeting the second goal (allowing users to easily develop a query) is to use as discriminators information we can automatically extract from from a structured design developed using a CASE tool. This allows module reuse at the design level; simply complete a structured design, then request modules that "best fit" that design. Extracting discriminators from existing modules can be done using reverse engineering tools. The remaining question is if the discriminators adequately characterize the semantics of modules. This paper describes a test of the ability of a given set of discriminators to perform this characterization of semantics.

We originally developed this idea in the area of heterogeneous databases. We extract discriminators describing an attribute in a database, and look for similar patterns in discriminators in other databases[7]. We have been successful in finding attributes that contain the same information based on similarities in these discriminators.

This paper presents a first step in applying this technique to software reuse. We evaluate the hypothesis that discriminators that can be obtained from a CASE-tool based design can be used to identify modules appropriate to that design. Specifically, we test if these discriminators can identify independently developed legacy modules that serve a common purpose. If so, we should be able to map the appropriate modules of one program into the design of the other.

The first step in this process is to extract factors describing attributes from the legacy modules (a vector of *discriminators* for each module). These vectors are used to train a back-propagation neural network[12]. The trained network can then be used to determine similarity with vectors extracted from a design. Figure 1 outlines this process.

One question is why do we use Neural Networks in this process? The discriminators provide a good deal of information to characterize modules. However, it is difficult to determine which discriminators will be helpful, and which are little more than "noise". Programmed computing is best used in those situations where the processing can be defined in terms of a known procedure or set of rules. We are dealing with a situation where the information used (the discriminators) are related to the semantics of the modules, but *how* they characterize those modules is not clearly known.

Neural networks have emerged as a powerful pattern recognition technique. They can perform tasks such as classification and generalization without being given rules since they are trained, not programmed. Neural networks

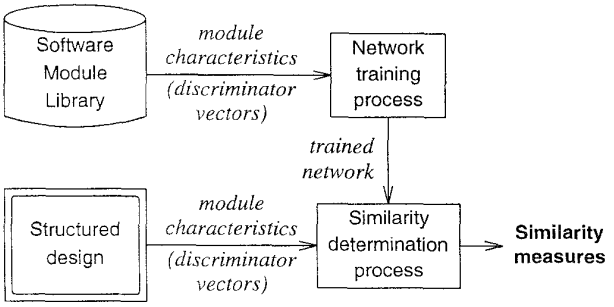

Figure 1: Automatic software module "query".

can also respond correctly to data not used in training. This is important, as we do not expect that reusable components to exactly match the design. We only hope to find candidate components that are close to our specifications, and we want to rank them based on their "closeness". We train the neural network to recognize how the discriminators capture the semantics (precisely, how they differentiate modules) for one collection of modules. This has two advantages:

- We avoid having to manually determine "rules" for how the discriminators capture semantics.
- We can develop a different set of "rules" for each library of modules.

We will first discuss related work in software indexing and reuse. In Section 3 we give specifics on the discriminators we use, and outline a test of their ability to characterize module semantics. Section 4 gives results of this test.

2. Related Work

The survey paper by Krueger[5] discussed different approaches for software reuse. Eight categories, such as high-level languages, source code components, application generators, etc., are discussed. In[3] the point was made that reuse will most likely succeed in narrow, well-understood application with slowly changing technologies, such as MIS or business systems (e.g., The Information Processing System Organization at Raytheon's missile Systems Division examined over 5000 COBOL source programs and identified only three major module classes: edit, update, and report.)

The effectiveness of a reuse technique can be measured in terms of *cognitive distance* – the effort required to use the technique. In[5], it was noted that cognitive distance can be reduced in two ways:

- Higher level abstractions in a reuse technique reduce the effort to go from the initial concept of a software component to representations in the reuse technique.
- Automation reduces the effort to map abstractions in a reuse technique to an executable implementation.

We are concerned with reuse of existing code, so we concentrate on improving automation at current levels of abstraction. There has been research in automatically

indexing and finding reuse candidates for code developed using existing programming methods. These can be divided into type-signature methods[13, 15] and keyword methods[4, 8, 9]. Type signatures are good for finding exact matches, but we also want to find cases where the reuse candidates are "close", and with some modification would serve the desired purpose. Keyword methods are effective for this purpose; however, they require consistency in word use so that keywords accurately reflect module semantics. This requires human effort and consistency, either on the part of the programmer (in choosing proper words), or an external librarian (who can build a dictionary of synonyms or more complex relationships between words). An example of an external librarian based method is faceted classification[11]. This was shown to be effective in small, domain-specific environments, but did not prove descriptive enough for a heterogeneous environment.

A third approach is given in[10]. Here the input/output behavior of the routines is used to determine semantic equivalence. This is like type signatures, in that it is good for finding exact matches. The disadvantage is the necessity of specifying the complete I/O behavior of the routine in advance; an extra burden on the designer.

One way to state the problem is that we need to find a set of factors that adequately characterize the semantics of a module. Our only constraints on these factors are that they:

1. Can be automatically extracted from the program.
2. Can be determined without actually writing the code (so as to make querying feasible).

Keywords are one such factor, type signatures are another. In this paper we discuss an approach using non-keyword information, as with type signatures, that gives a "closeness" measure (ranking), as with keyword methods. In addition, we do not predetermine exactly *how* the information we use describes the semantics; this is determined based on the programs in the "reuse database". We feel this will integrate well with keyword methods.

An example of a system that uses such collections of information is MCC's Domain Model/TAO, part of the DESIRE design recovery system[1, 2]. It assumes that Domain Model (problem, program, and application) knowledge can be used as patterns of informal and semi-formal information, or "Conceptual Abstractions". We follow this idea, in that we use a collection of information and a fuzzy matching process to recognize programs. However, we concentrate on automatically extractable knowledge and formalize the development of the fuzzy matching process. This gives an automated procedure, requiring very little human effort.

3. Test methodology

We must first test the hypothesis that we can create a set of discriminators that adequately characterize the semantics of the module. To do this, we asked a similar question: Can we use the discriminators to find modules from different programs that serve the same purpose? If so, we could instead use the *design* of one of the programs to generate discriminators and find appropriate *modules* from the other program. This allows us to test this method without any bias such as attempting to design to make use of known modules.

We now describe the specific details of this test. We will first discuss the discriminators used, and how they are obtained. We will then describe the programs used to generate the test "reuse library".

3.1. Discriminator choice

The first step is to extract discriminators (features we hope will capture the semantics of routines) from the programs. Keyword or natural language based methods do not map well into our process; besides, these methods have been studied elsewhere. Therefore we concentrate on non-keyword information that we could expect to be available at the design stage of a program. In particular, we looked at information that could be automatically extracted from a structured design represented in a CASE tool.

This meets our constraint that generating discriminators for a new module (to create a "query") be easier than writing the code (assuming the design is completed before the code). It also solves the problem of extracting discriminators from existing modules to place them in the library. Tools already exist to retrieve design information from existing code – this is reverse engineering. We use Cadre's Ensemble to reverse engineer programs into a structured design, and have written a program to extract discriminators from a structured design in Cadre's Team*work*. Thus we can automatically reverse engineer programs and extract discriminators from the design, giving the "indexing criteria" for existing code with no human effort. In addition, given that this information is present in the design while forward engineering we can just request code that fits our design – the "query" is automatically generated from the design itself.

The choice of discriminators is somewhat ad-hoc. We simply took all information we could extract from a reverse-engineered module that could be converted into numeric form that we felt might reasonably distinguish semantically related modules (such as type signatures). Such information has been used as part of a program understanding system[2]; thus we felt that it had potential to determine semantically related modules as well. The process given in Figure 1 automatically determines what information is and is not useful in discriminating between modules, so we simply provide all available information. The discriminators are shown in Table 1.[1]

[1] Note that some of this information would not be present in forward engineering, making query generation more difficult. To verify the *potential* of this method, we took all available information. In the last section we discuss how to eliminate some of these discriminators.

Table 1: Discriminators used in this test.

Discriminators 1-6: Number of parameters fitting into each of the following (exclusive) categories:

- Primitive
- Integer
- Real
- Character
- String
- Other (including user-defined)

Discriminators 7-10: Number of parameters fitting into each of the following (non-exclusive) categories:

- data in
- data out
- control in
- control out

Discriminators 11-20: Number of local variables used by the routine fitting into each of the above 10 categories.

Discriminators 21-30: Number of global variables used by the routine fitting into each of the above 10 categories.

Discriminators 31-34: The following information describing relationship to other routines (either in the design, or in the program from which the routine came):

- Total calls to the module
- Total number of routines making calls to the module
- Fan-out (calls to other routines by the module in question)
- Calls to library routines by the module

Discriminators 35-36: The following information describing the complexity of the routine:

- data complexity
- cyclomatic complexity

The pattern matching process requires a vector of values in the range [0..1]. Discriminators in Table 1 are all non-negative values; to map them to the desired range we compute a normalized discriminator for each value as follows:

$$normalized\ discriminator = 1 - \frac{1}{C^{discriminator}}$$

C is chosen to provide a reasonable range of normalized values near the estimated median for *value* (for example, $C \approx 1.12$ for normalizing the first 30 discriminators in Table 1. Thus routines with 0 vs. 6 integer parameters (mapping to 0 and ½) are considered roughly as "close" as those with 6 vs. a huge number of integer parameters (½ and 1).) This discounts extremely high values for the input value (for example character output routines may be called thousands of times in a large program and hundreds in a small program; they are still more likely to be similar than a routine called once and a routine called ten times).

3.2. Sample programs for reuse

In order to test this idea, we need "test data". We want a collection of programs that are likely to have semantically similar modules (routines that serve a similar purpose and thus would be good candidates for reuse), without having such similar routines be identical (if semantically similar routines are identical, the problem is too easy – not a good simulation of actual reuse scenarios). The X11 contributed clients serve this purpose well. Since many of these programs perform similar functions, there are many similar routines. However, the programs were written independently (and in fact contain complete programs with similar "specifications" that were developed completely independently). We expect that results for modules from programs designed in a common environment, as would be typical of software developed in a single corporation, would be better than this set; as similarities in design methods should lead to similar designs for modules with similar purposes.

We reverse-engineered each of 57 programs[2] into a structured design. This gave us 5056 routines to use as a sample reuse library. We then extracted the discriminators in Table 1 from each design for all the modules in that design.

We first checked to see if a sufficient number of the discriminator vectors were unique. Of the 5056 modules, 3490 had unique vectors. Of the non-unique ones, 254 were derived from identical routines found in exterm and kterm. Quick inspection of the remaining 401 duplicates showed many were quite similar in purpose (and thus would be good reuse candidates if we were independently developing one of the modules).

The next step is to determine if we can use this information to determine if routines are similar (in the sense that one could replace the other, or could be modified to replace the other with less effort than writing the routine from scratch.) To do this, we compare the discriminator vectors for one program with the vectors for all other programs. We look at two measures of success/failure:

$$\textbf{Recall}: \frac{\text{Reusable modules retrieved}}{\text{Reusable routines in the library}}$$

*How **complete** are the results.*

$$\textbf{Precision}: \frac{\text{Reusable modules retrieved}}{\text{Total modules retrieved}}$$

*How **correct** are the results.*

We will now discuss the specific tests we performed.

[2] The programs were chosen from the X11 contributed clients installed at in the Northwestern department of EECS, with the exception of some extremely large programs (as they exceeded the space we had available to reverse engineer them), and xakcl (a set of routines to map the AKCL lisp system to X – this consists of a large number of trivial, nearly identical routines).

4. Test results

To compute recall, it is necessary to know the expected answer, that is all of the similar routines (good reuse candidates) in the 5000 routines in the library. Due to the time required to compare over 5000 routines manually, we have not performed a complete test on all routines in the testbed. Instead, we performed a complete test on a subset of the routines, and determined precision figures for a larger set.

Section 4.1 describes the routines used in the complete test. In Section 4.2 we use a simple Euclidean distance metric to compare discriminators. Section 4.3 gives results using neural networks to find matches.

4.1. Complete test set

We studied three mail notification programs (xlbiff, xmailwatcher, and xpbiff) in detail. These are relatively compact (a total of 33 routines), written independently (so as not to skew the results because of actual copied routines), but likely to have similar modules due to having a similar purpose. We inspected these by hand; a quick overview is shown in Table 2. Note that we do not consider routines that can be replaced by a combination of other routines.

If we look at the number of "reusable modules", we find 20 (8 from xmailwatcher and xpbiff that could be used in writing xlbiff, 6 that could be used in xmailwatcher, and 6 for xpbiff). The size of the sample space (the total number of module pairs to be compared) is 688 (16 routines in xlbiff choose from the 9 in xmailwatcher and the 8 in xpbiff, 9·(16+8) for xmailwatcher, and 8·(16+9) for xpbiff).

4.2. Simple distance measurements

Our first test was to simply find the Euclidean distance between the vectors; the results (cut off at a distance of 0.4 out of a theoretical maximum of 6.) are shown in Table 3. Note that we found 4 of the possible 20 "matches" (pairs of similar routines), in 18 of the 688 possible pairs of routines. This gives a recall of 20% and a precision of 22%. This may appear low; however we must consider that this is achieved with very little *human* effort. Note that a simple routine name check would not find these pairs; this supports our conjecture that this would be a useful addition to keyword based methods.

An interesting comparison is how likely it would be to achieve this using random selection. The probability of having at least 4 good matches in 18 randomly selected pairs would be 1.2×10^{-3}. Another way of looking at this is the precision we would expect if we randomly selected pairs of routines. To get 4 good routines, we would expect to look at 131, giving a precision of 3% for random selection, as opposed to 20% using Euclidean distance of the discriminator vectors. Clearly, the discriminators are somehow descriptive of the semantics of the routines.

Another consideration is that a human looking through the five matches returned for xlbiff:handler and xmailwatcher:timedRescan would only need to look at the lowest distance match for each – after finding a good match, the remaining routines need not be inspected. This would improve to finding 4 in 15 trials, giving a precision of 27%.

We then tried the other "mail" programs in the test set (xmail and xmailtool). These are larger; a total of 236 routines. This gave us 175 matches (cutting off at 0.4). Of these, 7 are good candidates for reuse. Decreasing the cutoff to 0.3 retains 5 of the good matches, and cuts the total to 31. In practice, we could use the similarity rankings to determine the order to look at modules, this would improve the results. Table 4 shows the results if we stop after finding a good match – we need to look at 53 routines to find the 7 matches, or 15 to find 5 (using a cutoff of 0.3). Note that

Table 2: Modules in mail notification programs

xlbiff	*xmailwatcher*	*xpbiff*	*description*
CheckEvent	**eventHandler**		*Handle MapNotify, UnmapNotify events*
ErrExit			*Print error message and exit*
Exit			*Normal exit*
Popdown		**BreakPopup**	*Clear display*
Popup		**PopupMailHeader**	*Bring up display*
Shrink			*Handle simultaneous MapNotify and UnmapNotify*
Usage			*Give instructions*
checksize	**rescanMailbox**	**Polling**	*Check for and display new mail*
doScan	**parseMailbox**	**GetMailHeader**	*Get from/subject lines of new mail*
getDimensions			*Get size of text string*
handler	**timedRescan**		*Reschedule check of mailbox file*
initStaticData			*Initialization routine*
lbiffRealize			*Fill window with text and bring it up*
lbiffUnrealize			*Kill window*
toggle_key_led			*Turn on/off keyboard led*
	beep		*Makes a sound*
	handlePrevious		*Skip old message*
	mungeSender		*Get the message sender*
	setTitle		*Set icon/window title*
		AnimateBiff	*Display new graphic*
		popup_again	*Check for new mail on request*
		redraw_callback	*Redraw the display*
main			*Outermost routine*
	main		*Outermost routine*
		main	*Outermost routine*

Note: *Similar routines (those that could feasibly be used to replace each other with less work than writing from scratch) are shown on the same line and boldfaced.*

Table 3: Euclidean distance between routines in mail notification programs.

Source module	*Modules with distance < 0.4*
xlbiff:Exit	0.318: xmailwatcher:eventHandler
	0.363: xmailwatcher:timedRescan
xlbiff:Popup	0.385: xmailwatcher:beep
xlbiff:Usage	0.298: xmailwatcher:beep
xlbiff:doScan	**0.336: xpbiff:GetMailHeader**
xlbiff:handler	**0.: xmailwatcher:timedRescan**
	0.359: xmailwatcher:beep
xlbiff:lbiffUnrealize	0.318: xmailwatcher:beep
xmailwatcher:beep	0.298: xlbiff:Usage
	0.318: xlbiff:lbiffUnrealize
	0.359: xlbiff:handler
	0.385: xlbiff:Popup
xmailwatcher:eventHandler	0.318: xlbiff:Exit
xmailwatcher:timedRescan	**0.: xlbiff:handler**
	0.363: xlbiff:Exit
	0.369: xlbiff:lbiffUnrealize
xpbiff:GetMailHeader	**0.336: xlbiff:doScan**

Note: *Routines that are actually semantically similar (reuse candidates) boldfaced.*

our precision is still in the range of the smaller test (13-33%, depending on the choice of cutoff). We are not able to state the recall with the same degree of confidence, however based on the known available matches, plus an additional 5 found in an inspection of xmail and xmailtool, we can estimate the recall as in the neighborhood of 20%. Note that there were seven routines for which no good match was found (e.g., xlbiff:Popdown); a total of 36 potential candidates were returned for these.

However, the real question is does this scale well. We expanded to look at all 5056 routines in the test set, with a cutoff of 0.2. In addition to the 3 good matches already shown with distance less than 0.2, we found multiple matches for xlbiff:Usage (we also found a second match for xlbiff:Quit). Using the metric above (looking through all returned routines until we have either exhausted the set, or found a good match), we must look through 21 routines in order to find our good ones, a 29% precision. Of these, 12 are due to xmailwatcher:beep; if we had also used a name-based screen we would have found a match for this quickly (and probably for xlbiff:Exit and xlbiff:Usage as well). However, we still have matches that would *not* be obvious from the routine name.

4.3. Using neural networks for matching

Using Euclidean distance as a metric fails to account for differences in the relative importance of different discriminators. To determine this importance manually would be difficult, however we can use the technique from[7] to come up with a reasonable solution automatically. We train a neural network to recognize the individual routines based on their discriminators. The training phase will determine how the given discriminators best distinguish between the routines *in the training set*. We can then use this network to find how a different set of discriminators (a "query") relates with those in the training set. Using a library of components as our training set should give us a good way to search for the close matches *among those specific components*.

Table 4: Euclidean distance between mail notification routines and other mail program routines.

Source module	*First match or distance < 0.4*
xlbiff:CheckEvent	0.292: xmailtool:InParams
	0.337: xmailtool:XMTlabel
	0.341: xmail:info_handler
	0.358: xmailtool:map_handler
xlbiff:Exit	**0.179: xmail:Quit**
xlbiff:Popdown	0.319: xmailtool:delete_msg
	4 others, no matches
xlbiff:Popup	0.293: xmailtool:update_mbox_icon
	0.295: xmailtool:undelete_msg
	0.315: xmailtool:delete_msg
	5 others, no matches
xlbiff:Shrink	0.380: xmail:info_handler
	0.390: xmail:ShowHelp
xlbiff:Usage	0.230: xmailtool:free_aliases
	0.247: xmailtool:park_mail
	0.268: xmailtool:Syntax
xlbiff:checksize	0.358: xmail:file_handler
xlbiff:handler	**0.226: xmailtool:mail_timeout**
xlbiff:lbiffUnrealize	0.290: xmailtool:pr_msg_list
	0.301: xmailtool:free_headers
	0.328: xmailtool:update_mbox_icon
	6 others, no matches
xlbiff:toggle_key_led	0.389: xmail:figureWidth
	0.398: xmailtool:pr_msg_list
xmailwatcher:beep	0.189: xmailtool:free_headers
	0.231: xmailtool:free_aliases
	0.276: xmailtool:beep
xmailwatcher:eventHandler	0.244: xmail:Quit
	0.266: xmail:info_handler
	0.354: xmailtool:XMTWMProtocols
	0.368: xmailtool:map_handler
xmailwatcher:timedRescan	**0.226: xmailtool:mail_timeout**
xpbiff:BreakPopup	0.346: xmail:Iconify
	8 others, no matches

Note: *Routines that are actually semantically similar (reuse candidates) boldfaced.*

Each of the three programs is compared with a net trained using the routines of the other two programs. (If we were to train a net with all 36 routines, it would invariably note that each routine matches only itself.) A higher number represents a closer match (we have eliminated all matches below 0.5). The results are shown in Table 5.

This gives us 6 good matches in 27 returned pairs This is a recall of 30% and a precision of 22%. On the one hand, this is not substantially better than simple Euclidean distance. However, achieving this with random selection would only happen with probability 5.5×10^{-5}. In addition, the "fixed threshold" that we have used isn't appropriate for the networks – the value that will be returned for a good match varies for different networks. If we instead choose an individual threshold for each network (0.8 for the first two,

Table 5: Similarity of routines in mail notification programs.

Source module	*Modules with similarity > 0.5*
xlbiff:Exit	0.875: xmailwatcher:timedRescan
	0.764: xmailwatcher:eventHandler
xlbiff:Shrink	0.750: xpbiff:popup_again
	0.684: xmailwatcher:eventHandler
	0.543: xmailwatcher:timedRescan
xlbiff:Usage	0.697: xmailwatcher:beep
xlbiff:checksize	**0.846: xpbiff:Polling**
	0.779: xpbiff:GetMailHeader
xlbiff:doScan	**0.957: xpbiff:GetMailHeader**
xlbiff:getDimensions	0.873: xmailwatcher:mungeSender
xlbiff:handler	**0.916: xmailwatcher:timedRescan**
xlbiff:main	0.912: xmailwatcher:main
xmailwatcher:beep	0.744: xlbiff:Usage
	0.658: xlbiff:lbiffUnrealize
	0.547: xlbiff:handler
xmailwatcher:main	0.509: xlbiff:main
xmailwatcher:timedRescan	**0.947: xlbiff:handler**
xpbiff:AnimateBiff	0.819: xmailwatcher:main
	0.571: xlbiff:toggle_key_led
xpbiff:GetMailHeader	**0.966: xlbiff:doScan**
xpbiff:Polling	0.929: xlbiff:Popdown
	0.598: xlbiff:checksize
xpbiff:PopupMailHeader	0.984: xmailwatcher:main
	0.507: xmailwatcher:setTitle
xpbiff:popup_again	0.803: xlbiff:Shrink
xpbiff:redraw_callback	0.955: xmailwatcher:main
	0.521: xmailwatcher:setTitle

Note: Routines that are actually semantically similar (reuse candidates) boldfaced.

and 0.5 for the last) we find 6 in 17, an increase to 35% precision. Another advantage to this method is that the good matches are clustered near the top of the range. If we simply take all matches with a similarity measure ≥ 0.8, we find 5 matches in 13 (probability 1.4×10^{-5}). If we use a threshold of 0.9, we find 4 matches in 8 (a 50% precision, even though recall has dropped to 20% – achieving this through random selection would only happen with probability 3.4×10^{-5}). Euclidean distance does not share this property – to get 4 matches, we have to inspect at least 10 pairs (and this requires choosing a cutoff between 0.337 and 0.358 – an unlikely choice without a-priori knowledge of the results).

Table 6: Similarity between each mail notification routine and all other mail program's routines.

Source module	*Modules with similarity > 0.8*
xlbiff:Exit	**1.000: xmail:Quit**
xlbiff:getDimensions	0.943: xmailtool:save_proc
xlbiff:handler	**0.989: xmailwatcher:timedRescan**
xlbiff:initStaticData	0.997: xmailtool:Syntax
xmailwatcher:eventHandler	0.966: xmail:info_handler
xmailwatcher:main	0.972: xmailtool:confirm_send
xmailwatcher:timedRescan	**0.983: xlbiff:handler**
xpbiff:redraw_callback	0.954: xmailtool:center_wig_on_pointer

Note: Routines that are actually semantically similar (reuse candidates) boldfaced.

We have also looked for matches between these 33 routines and the larger set including xmail and xmailtool. This was done by training a network with all of the routines *except* those from one program, then finding similarities to the routines in that program. This results in three somewhat separate tests, summarized in Table 6. The precision here is 38%. Again, we are not able to say the actual recall, but based on likely matches we have been able to find (25), we estimate the recall to be around 12%.

One difficulty with the small test set is that it is "too easy" for the neural network to come up with a function distinguishing the 33 routines based on the 36 available discriminators. This is why the precision is comparable for the larger test. As an example, if we use the network based on the 33 routines to compare with the full set 5056 X11 client routines, we have a large number of matches (1160) with a threshold of 0.8 (raising the threshold to 0.9 only drops this to 822). However, if we train a network to recognize all 269 mail routines, then use this to search for similar routines in the 5056 routine set (with the same threshold), we find (with a threshold of 0.8), we find 89 for xpbiff, 196 for xlbiff, and 65 for xmailwatcher. This shows that this idea scales well; searches over larger libraries are automatically more selective.

5. Conclusions and Further Work

We have shown that we can often match similar routines without using either human effort to classify the routines, semantic information contained in names, or any sophisticated semantic analysis of the code. Due to the use of information that is either likely to be present in the design, or can be estimated (such as complexity), it is likely that the discriminators we have used could be obtained with substantially less effort than writing the code. This matches our criteria for a good method for classifying a "legacy component library":

- No effort is required to insert a routine into the reuse library.
- Finding reuse candidates is a low-effort part of the existing software development process.

The results should be orthogonal to name based methods. Straightforward module name lookup (combined with a keyword/synonym dictionary) meets the above criteria, and more advanced methods (requiring greater human effort) have been explored. Combining these methods should be easy and effective. This would lead to two things: "serendipitous" finds (where names/keywords would not be helpful), and additional help in ranking potential modules.

One disadvantage of this technique is that it only works if modules and designs with similar semantics have similar discriminators. This requires consistency in design. However, the same problem appears in keyword based techniques or a human-classified library. These work because the choice of words or human description of a routine

happens to match the actual semantics. We are simply looking at *design decisions* as opposed to word choices as a "human description" that reflects the semantics of a module. The interesting result is that it *does* work (provide added value), and does so without any extra *human* cost. In addition, *how* it works can vary depending on the modules in the "reuse base"; the neural network is trained to recognize how the discriminators characterize *the given set of modules*.

Further work

We are pursuing experiments to learn more about *how* and *why* this works. In particular, we want to determine what discriminators are most useful: If, for example, cyclomatic complexity is not helpful in finding modules, we don't want to ask the user of a legacy software library to provide a complexity estimate. We are looking at ways of determining this; one method is simply to try dropping discriminators, re-run the experiments, and compare the results with the full set.

Once we have determined an appropriate set of discriminators, we can begin to define a "query language" for finding modules. In the best case, this will be integrated into the design process, so that the "query" can be automatically determined from design information. We can then combine this with other methods for finding reusable components.

A second difficulty is *training* large networks. We are able to train networks for hundreds of modules (the time being on the order of hours or days on a workstation), however training networks for thousands of modules in a reasonable time will require substantially improved training algorithms or special-purpose hardware. One solution is to gather code into groups of a few hundred modules, and train separate networks for each. This will give poorer precision than a single net, but can improve recall.

The most important question, however, is can this be useful in practice? One of the main strong points of this method is the low human effort required – a practical test can be made by incorporating this within a comprehensive design/reuse environment. This requires choosing discriminators that can be derived from both new designs and legacy code. Users can then try this method with no change in the design/coding process, and decide for themselves if the technique has value.

References

1. Ted J. Biggerstaff, "Human-Oriented Conceptual Abstractions in the Re-engineering of Software," pp. 120 in *Proceedings in the 12th International Conference on Software Engineering*, IEEE, (1990).
2. Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster, "Program Understanding and the Concept Assignment Problem," *Communications of the ACM* **37**(5) pp. 72-83 ACM, (May 1994).
3. William B. Franks, Ted J. Biggerstaff, Kazuo Matsumura, Ruben Prieto-Diaz, and Wilhelm Schaefer, "Software Reuse: Is it Delivering?," pp. 52-59 in *Proceedings in the 13th International Conference on Software Engineering*, IEEE, (1991).
4. M. R. Girardi and B. Ibrahim, "A Similarity Measure for Retrieving Software Artifacts," in *Proceedings of the 3rd International Conference on Software Reuse (ICSR '94)*, Rio de Janeiro, Brazil (November 1-4, 1994).
5. Charles W. Krueger, "Software Reuse," *Computer Surveys* **24**(2) pp. 131-184 ACM, (June 1992).
6. David Lauzon and Thomas Rose, "Task Oriented and Similarity-Based Retrieval," in *Proceedings of the 9th conference on Knowledge-Based Software Engineering*, IEEE, (September 21-23, 1994).
7. Wen-Syan Li and Chris Clifton, "Semantic Integration in Heterogeneous Databases Using Neural Networks," pp. 1-12 in *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile (September 12-15, 1994).
8. Y. Maarek, D. Berry, and G. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *Transactions on Software Engineering* **SE-17**(8) pp. 800-813 IEEE, (August 1991).
9. Dieter Merkl and A Min Tjoa, "Retrieval of Reusable Software Based on Self-Organizing Feature Maps," in *Proceedings of the 6th Int'l Conference on Artificial Intelligence and Expert Systems Applications (EXPERSYS-94)*, Houston, TX (Dec. 1-2. 1994).
10. Andy Podgurski and Lynn Pierce, "Retrieving Reusable Software by Sampling Behavior," *Transactions on Software Engineering and Methodology* **2**(3) pp. 286-303 ACM, (July 1993).
11. Rubén Prieto-Díaz, "Implementing Faceted Classification for Software Reuse," *Communications of the ACM* **34**(5) pp. 88-97 (May 1991).
12. David E. Rumelhart, Bernard Widrow, and Michael A. Lehr, "The Basic Ideas in Neural Networks," *Communications of the ACM* **37**(3) pp. 87-92 ACM, (March 1994).
13. Colin Runciman and Ian Toyn, "Retrieving re-usable software components by polymorphic type," pp. 166-173 in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, Imperial College, London (September 11-13, 1989).
14. Will Tracz, "Where Does Reuse Start?," *Software Engineering Notes* **15**(2) pp. 42-46 (April 1990).
15. A. M. Zaremski and J. M. Wing, "Signature Matching: A Key to Reuse," in *Proceedings of SIGSOFT*, ACM,Los Angeles, California (December 7-10 1993).