Assessing Software Libraries by Browsing Similar Classes, Functions, and Relationships

Technical Report UW-CSE-98-08-05

Amir Michail and David Notkin
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA
{amir,notkin}@cs.washington.edu

ABSTRACT
Comparing and contrasting a set of software libraries is useful for reuse related activities such as selecting a library from among several candidates or porting an application from one library to another. The current state of the art in assessing libraries relies on qualitative methods. In particular, the developer manually inspects each library, reads the documentation, examines the architecture, considers subjective scenarios, and other available information.

To reduce costs and/or assess a large collection of libraries, automation is necessary. Although there are tools that help a developer examine an individual library in terms of architecture, style, etc., we know of no tools that help the developer directly compare several libraries. With existing tools, the user must manually integrate the knowledge learned about each library.

Automation to help developers directly compare and contrast libraries requires matching of similar components (such as classes and functions) across libraries and various relationships (such as inheritance and invocation) between them. This is different than the traditional component retrieval problem in which components are returned that best match a user's query. Rather, we need to find those components and relationships that are similar across the libraries under consideration. In this paper, we show how this kind of matching can be done.

Keywords
Software libraries, reuse, assessment, information retrieval

1 INTRODUCTION

Comparing and contrasting a set of software libraries is useful for reuse related activities such as selecting a library from among several candidates or porting an application from one library to another. Library selection in particular is difficult and can be very expensive. Indeed, Sparks, Benner and Faris give this advice for framework selection:

> Budget adequately to support frameworks. Expect the evaluation and selection of a framework to take up to six staff-months per new framework. [16, p. 54]

The current state of the art in selecting among library candidates relies on qualitative assessment. This may take the form of informal tips for selecting frameworks [16] or a complete analysis method, such as SAAM [9]. Either way, the developer manually inspects each library, reads the documentation, examines the architecture, considers subjective scenarios, and other available information.

To reduce costs and/or assess a large collection of libraries, automation is necessary. Although there are tools that help a developer examine an individual library in terms of architecture, style, etc. [1, 10, 13, 18], we know of no tools that help the developer directly compare several libraries. With existing tools, the user must manually integrate the knowledge learned about each library.

Automation to help developers directly compare and contrast libraries requires matching of similar components (such as classes and functions) across libraries and various relationships (such as inheritance and invocation) between them. This is different than the traditional component retrieval problem in which components are returned that best match a user's query. In our case, there is no user query per se. Rather, we need to find those components and relationships that are similar across the libraries under consideration. In this paper, we show how this kind of matching can be done.

Specifically, we present two matching techniques, *name matching* and *similarity matching*. The name matching method matches those components that have the same "standardized name" in each library. The similarity matching method uses more conventional information retrieval techniques and is similar to that used in component retrieval tools based on free-text indexing [5, 8, 11].

In this paper we shall be concerned with components that are classes or functions. To simplify the exposition, we will just say "component" whenever the discussion applies to both classes and functions. To avoid ambiguity, keep in mind that we will only match classes with classes and functions with

functions; we never match a class with a function.

The paper is organized as follows. Section 2 discusses related work. Section 3 describes the components and the relationships between them that we extract from the source. Section 4 presents the name matching technique. Section 5 shows the similarity matching method. Section 6 presents experimental results. Section 7 discusses a tool based on these techniques and shows, by example, how it can be used to assess libraries. Section 8 summarizes the work, concluding with a number of open questions.

2 RELATED WORK

As mentioned earlier, we know of no tools that help a developer directly compare several libraries. With existing tools [1, 10, 13, 18], the developer must manually integrate the knowledge learned about each library. Yet, the problem of finding similar components across libraries appears equivalent to the traditional component retrieval problem in which components are returned that best match a user's query. This is only partially true.

The whole point of component retrieval is that the user needs a component that performs some function yet does not know the *exact* name of that component (or even if it exists in the library). However, when comparing components across libraries in a particular domain, it is likely that the developers of each library are themselves experts in the domain and have chosen standard terminology to name components — at least for those that are fundamental to that domain. For this reason, it makes sense to match components by name across libraries (after some initial preprocessing such as stripping the library prefix, if any), which is exactly what we do in Section 4.

But name matching alone is not enough. It is still possible that some important concept is represented by components with completely different names in the libraries. For this reason, we consider well-known component retrieval techniques such as free-text indexing [5], facets [14], and formal specifications [2]. However, since we are interested in automated techniques that do not require domain analysis or formal specifications in the code, we exclude facets and formal specifications from our discussion.

The free-text indexing method simply uses the text in the libraries (and not just the component names) for indexing using standard information retrieval techniques [4]. No manual domain analysis is required. However, researchers have observed that this method works well with libraries that include extensive documentation with the components, such as Unix man pages [6, 11]. Only then can one rely on regularities in the text such as relative word frequencies or lexical affinities [11]. However, it has also been suggested that one can combine library documentation with structural information that can be extracted from the source such as inheritance relationships [8].

Rather than restrict the kinds of libraries that can be compared and contrasted, we have decided not to rely on component documentation. Consequently, we do not look for regularities in the text; rather, we define a similarity measure that makes heavy use of structural information (and in a more extensive manner than [8]). We describe our similarity measure in Section 5.

3 INFORMATION EXTRACTION

In this section, we describe the components and relationships that we extract from the library source code. This extraction process is the only language dependent aspect of our approach. (Our tool currently handles C, C++, and Java.)

Components

As mentioned earlier, a component is either a class or a function. We extract any type as a class, whether it appears in the source as a struct, class, interface, or union. We also extract all functions from the source, whether they are members of a class or not. Moreover, we consider class member variables as functions. (In essence, a member variable is a function that takes one argument containing a pointer to the instance of the class.)

Relationships

We also extract various relationships between components. These may occur between two classes, as with inheritance and composition, between two functions, as with invocation, or between a class and a function, as with association. We extract these relationships from the source not only for the purpose of matching across libraries but also because they are used in computing the similarity measure for similarity matching.

Inheritance and Composition

The two most common techniques for reuse in object-oriented libraries are class inheritance and composition [7, p. 18]. When using inheritance, a class A "inherits" from some class B. When using composition, a class A has B as one of its member variables. (However, not all member variables indicate composition; some only express acquaintance relationships with other objects in the system.)

Now it may be that one library uses inheritance while another uses composition to express a reuse of B in A. To capture a similarity even in this case, we view inheritance and composition as forms of a *reuse relationship*; thus, in both libraries, A reuses B.

Finally, to further increase the likelihood that some relationships match across libraries, we not only consider direct reuse relationships but also indirect ones. For example, it may be the case that PushButton doesn't directly inherit from Widget in every library but that it does inherit directly/indirectly in each of the libraries.

As a practical note, observe that composition is not a syntactic construct in many languages, so we need to distinguish it from the weaker acquaintance relationship. In a language such as C++, developers may use member variables that are

real instances rather than pointers or references to instances. We consider all such uses as composition.

However, pointers or references to instances may also be used to indicate composition (and indeed, it is the only way in some languages such as Java). In such cases, we use the following heuristic: we consider a pointer/reference to class B in A to be composition if we can find code in the class members of A that allocates new instances of B.

Invocation
The invocation relationship indicates a call from one function to another. As with the inheritance, composition, and reuse relationships, we not only look for direct relationships but indirect ones as well.

However, the indirect invocation is problematic in the following sense: does function f indirectly invoke h when f invokes g and g invokes h? It may be the case that no sequence of calls from f to g also ends up invoking g even though in other situations, g does call h.

To avoid this (undecidable) problem, we simply say that f indirectly invokes h if there is some sequence of direct invocations $f, g_1, \cdots, g_k, h$. (It may be that no call sequence actually occurs that goes from f to h; this depends on the logic of the code.)

Association
In object-oriented languages, classes contain member functions. If class C contains a member function f, then we say that f is an *associated function* of C, or equivalently, that C is an *associated class* of f. The relationship always goes both ways.

In languages that are not object-oriented, there is no notion of a class with member functions. However, recall that we make no distinction between member variables and member functions so any variables declared in a structure are associated with that structure.

Moreover, it is common practice for developers to associate a function with a structure by supplying (a pointer to) the instance of that structure as the first parameter to the function. By looking at the type of the first parameter, we can infer associations between functions and the structures that they operate upon.

4 NAME MATCHING

A component with the same name in different libraries is likely to serve a similar purpose in each of those libraries. Yet, it is unlikely that one would find a component with *exactly* the same name in several libraries due to different naming conventions.

For example, words in a component name may be separated by underscores, changes in case, or some combination of the two. Moreover, developers often prefix component names with a distinct library prefix to prevent name clashes with other libraries and the application code.

In this section, we show how to standardize component names in each library, and then we show how to match classes and functions across libraries based on the standardized names.

Standardizing Names
There are two basic steps in standardizing names: (1) we identify the words in each name; and (2) we remove non-essential words. The words are then appended to form a new name where each word starts with an uppercase letter.

Identifying Words
The first step in standardizing a component name is to split it up into a sequence of words. We infer word boundaries from underscores and/or changes in case. While inferring word boundaries from underscores is straightforward, identifying words from case change is more involved.

In particular, a transition from a lower case letter to an upper case letter signals the start of a new word. Moreover, a sequence of uppercase letters ending the name or followed by an underscore constitute a single word. For example, JX_window is separated into "jx", "window". If a sequence of uppercase letters is followed by a lower case letter, then this is split into a (possibly empty) word that includes all the uppercase letters except the last and another word that starts with the last uppercase letter and any subsequent lowercase letters. For example, CScrollBar is separated into the sequence of words "c", "scroll", "bar", while JX_ScrollbarClass is separated into "jx", "scrollbar", "class".

Although not good style, some developers may not separate words at all in a component name. Moreover, it is not always clear whether a concept is written as one or two words, as with "Scrollbar" versus "ScrollBar". Some libraries may use one variation while others use the other.

For this reason, after having done the steps described above, we attempt to break the individual word strings even further into two or three English words using dictionary lookup. For example, the word string "scrollbar" is broken up into the words "scroll" and "bar". (In cases of ambiguity, we separate the string into as few words as possible, with shorter words appearing near the beginning of the sequence.)

We also make use of a table of common abbreviations, which helps us avoid problems with trying to match an abbreviated word in one library with an unabbreviated one in another. (Currently, this is only done for some well-known domain independent abbreviations; it is possible to include domain dependent ones also, but this would require domain analysis.) Abbreviation expansion is also used while breaking up word strings. For example, the string "appcmd" is broken up into the sequence "application", "command".

Removing Non-Essential Words
As mentioned earlier, developers often prefix component names with a distinct library prefix to prevent name clashes

with other libraries and the application code. This is particularly true with languages without name space management such as C and early versions of C++. Removing library prefixes is important since we may otherwise miss matching a class, say JXWindow, in one library, with one in another library, say QWindow, since they have different prefixes.

A library may have one or more prefixes, and a prefix may contain one or more words. We identify library prefixes simply by looking for prefixes that occur frequently. It is not likely that a prefix contributes meaningfully to a name if it occurs in many other names. The technique we use is identical for classes and functions and is done separately for each independent of the other.

In particular, performing the steps described above yields a sequence of words for each component name in the library. For each such sequence, we remove the *maximal prefix of words*, each at most l letters long, that occurs at least k times as a prefix of components in the library. In practice, we find $l = 3$ and $k = \min(10, 0.1N)$ yields good results, where N is the number of components in the library.

For example, this procedure would remove the "c" prefix in "c", "scroll", "bar", and the "jx" prefix in "jx", "scroll", "bar", "class".

Once the library prefix(es) are removed, we also remove any standard prefixes and suffixes in the remaining words in the component name. These include such standard function prefixes as "get", "set", "is", and standard class suffices such as "class", "info", "type". For example, we would remove the "class" suffix from the word sequence "scroll", "bar", "class".

In languages that are not object-oriented, developers may none-the-less write code in an object-oriented style and indicate that a function is a member of class by embedding the class name in the function name. For example, the name widget_show indicates that this function defines the show member of the widget class. It is also standard practice to supply a pointer to the widget structure as the first parameter in the widget_show function.

In such cases, we remove the embedded class name for the purpose of matching across libraries (some of which may be written in object-oriented languages where this practice is not required nor used). We do this by first standardizing the class names and then checking whether the type of the first parameter of a function has a name that is embedded in the function name. If so, we remove the embedded class name from the function name.

Matching across Libraries

We match all components that have the same standardized name in each library under consideration. However, observe that the standardization process may map two different names to the same name.

For example, a library may have classes JXTextEditor and JTextEditor, both of whose names get mapped to TextEditor (since JX and J are both library prefixes). Similarly, two functions getTextColor and setTextColor will have their names both mapped to TextColor.

To address this issue, we simply make all such components part of the same *component family*, whose name is the standardized name of its members. For example, JXTextEditor and JTextEditor are both members of the TextEditor class family. Similarly, getTextColor and setTextColor are members of the TextColor function family.

Thus, we actually match component families across libraries. However, we also remember the members of each family. In this way, a developer may see which class and function families a set of libraries share, and at the same time, be able to inspect the members of a family in each library.

Finally, we also need to define what we mean by relationships between families (such as those described in Section 3). A relationship between two families A and B holds if and only if that relationship holds for at least two members, with one member from each of the two families.

For example, classes JTextEditor and JXTextEditor both belong to the same class family, TextEditor. Even though only class JTextEditor contains a member function Paste, we still have an "association" relationship between the TextEditor family and the Paste function family.

To simplify the exposition in the remainder of the paper, we simply say "class" and "function" to mean "class family" and "function family", respectively.

5 SIMILARITY MATCHING

It may be the case that similar classes or functions in different libraries have names that are not matched by the name matching technique described in Section 4. In this section, we describe a complementary technique, similarity matching, that not only takes into account the class (function) name, but also its associated functions (associated classes) and related comments.

In what follows, we assume that the name standardization technique described in Section 4 has been carried out. And as mentioned earlier, we simply say "class" and "function" to mean "class family" and "function family", respectively.

We define a *similarity measure* that indicates how closely two components are related. The ideas in this section borrow heavily from the field of information retrieval [4], where similarity measures are used to rank documents returned by a query in order of relevance. In our case, there is no query per se, but we view components as documents D_i and compute the similarity between components in one library with components in another.

We associate with each component D_i a set of terms T_i that are extracted automatically from the source code. Some

terms have more weight than others, and we use $w_i(t)$ to denote the weight of term t in component D_i. Given two components D_i and D_j in different libraries, we define the similarity between them, $S_{i,j}$, as the "dot product" of the weights [17]:

$$S_{i,j} = \sum_{t \in T_i \cap T_j} w_i(t) \cdot w_j(t).$$

(We do not use a "normalized" similarity function, such as the cosine coefficient where the expression above is divided by $\sqrt{\sum_{t \in T_i} w_i^2(t)} \sqrt{\sum_{t \in T_j} w_j^2(t)}$ [17], because the components usually don't have enough terms associated with them for this to yield better results.)

The remainder of this section shows how to extract terms and determine the associated weights for each component in a library. This is done independently of any other libraries.

Extracting Terms

Terms for a component are extracted from three sources of information in the code: the class (function) name, its associated functions (associated classes), and related comments. In all three cases, we expand any common abbreviations, such as "cmd" for "command" and "len" for "length", and put words in their base forms, as with "run" for "running" and "directory" for "directories".

We also use a *stop list* to filter out words that tend not to help in distinguishing a component from another. These include *closed-class* words — pronouns, prepositions, conjunctions, and interjections — as well as commonly used terminology such as "copy", "initialize", and "iterate".

Name

A component name tends to contain one or more words that are usually very good indicators of the purpose of that component. Although the component name may not match in its entirety with one in another library, the word(s) in the name are extracted as terms for the purpose of similarity matching.

Associations

The associated functions of a class provide information about the purpose of that class. Similarly, the associated classes of a function provide information that may help "narrow down" the purpose of that function. For these reasons, we extract terms from certain "inherently" associated components as discussed in what follows.

Not all associations give us information that is inherent to a component. In particular, a class C may inherit or override a member function f from an ancestor in the inheritance hierarchy. In such a case, we do not expect the information provided by f to be as inherent to class C as that from another member g that is defined in C but not present in any of its ancestors.

We can make a similar observation concerning *delegation*. Delegation is often used to make composition as powerful for reuse as inheritance. This is done by having a class C "delegate" a call to one of its member functions f to another function (usually of the same name) in one of its instance variables. (This is analogous to a class deferring a request to one of its parents using inheritance.) Again, in such a case, f is not inherent to C.

We say that f is an *inherently associated function* of C, or equivalently, that C is an *inherently associated class* of f, if and only if no ancestor of C defines f and no call to f is "delegated" to another member function f in one of C's instance variables. We extract terms for a class (function) only from word(s) in the names of its inherently associated functions (inherently associated classes).

Comments

Finally, we extract terms from component comments. We try to include comments that describe the comment's functionality but not its implementation. This is done by looking for comments at the beginning of a component or those that immediately follow its declaration without an intervening newline (but we omit those comments buried in the definition of the component body).

Although we include the comments for all classes, we do not include the comments for all functions. In particular, if function f is associated with class C but not in an inherent manner, then we ignore any comments concerning f in class C.

Computing Weights

Now that we have described how terms are extracted from the source code for components, we show how to compute the weight $w_i(t)$ for each term t associated with component D_i. It is standard practice to define each weight as the product of the *inverse document frequency* and the *within-document frequency* [15].

However, as mentioned earlier, we cannot rely on term frequencies since the "documents" in our case are short. For this reason, we do not use the within-document frequency. Instead, we rely on structural information which is supplied as the *within-document weight*.

Thus, we define each weight $w_i(t)$ as the product of the *inverse document frequency*, $idf(t)$, and the *within-document weight*, $wdw_i(t)$,

$$w_i(t) = idf(t) \cdot wdw_i(t).$$

Inverse Document Frequency

The inverse document frequency $idf(t)$ indicates how "important" term t is in the library. If t occurs frequently in many components, then it is not a good discriminator and should not be weighed heavily. If t is quite rare, then it is likely to yield more information and should have greater weight.

Let N denote the total number of components in the library, and let $df(t)$ denote the number of components containing

term t. We use the following definition for $idf(t)$, which is decreasing in $df(t)$, as proposed in [3]:

$$idf(t) = \log_2 \left(\frac{N}{df(t)} - 1 \right).$$

Within-Document Weight
The within-document weight $wdw_i(t)$ indicates how "important" term t is in a particular component. We compute $wdw_i(t)$ by summing direct contributions $dc_i(t)$ from terms associated with component D_i and indirect contributions $ic_i(t)$ obtained by considering closely related components.

First, we calculate $dc_i(t)$. Recall that the terms for class (function) D_i are extracted from three sources of information: the name, inherently associated functions (classes), and related comments. A term may come from one, two, or all three sources. Let $a_1(i,t)$, $a_2(i,t)$, and $a_3(i,t)$ denote the weights for the three sources — name, inherent associations, and comments, respectively — for a term t in component D_i. The direct contribution $dc_i(t)$ for term t's weight in component D_i is:

$$dc_i(t) = a_1(i,t) + a_2(i,t) + a_3(i,t).$$

Each $a_k(i,t)$ is defined as follows:

$$a_k(i,t) = \begin{cases} A_k / 2^{b_k(i,t)-1} & \text{if term } t \text{ is in source } k \\ 0 & \text{otherwise} \end{cases}$$

where A_k is a constant indicating the importance of source k, and $b_k(i,t)$ indicates the minimum number of words in an identifier/word that contains term t in source k. For example, the term "dialog" is weighed twice as heavily if it is obtained from Dialog than if it were obtained from FileDialog (and there is no other identifier/word "Dialog"). In practice, we find that $A_1 = 2$, $A_2 = 4$, $A_3 = 1$ yield good results for both class and function matching. (We use these values in the experiments in Section 6.)

Now that we have shown how to compute the direct contribution $dc_i(t)$ for the within-document weight $wdw_i(t)$, we now consider the indirect contribution $ic_i(t)$ which considers terms associated with closely related components in the same library.

For class matching, consider the graph G whose nodes consist of classes in the library and whose edges denote inheritance and/or composition relationships between them. Let $d(D_i, D_j)$ be the length of the shortest path from D_i to D_j in G. Then the indirect contribution $ic_i(t)$ from other classes to class D_i is:

$$ic_i(t) = \sum_{D_j \in R_i} dc_j(t) / 2^{d(D_i, D_j)+1}$$

Domain	Library	Language	Classes	Functions
GUI	Qt-1.40	C++	301	1285
	JX-1.0.8	C++	228	740
	Kaffe-1.0	Java	128	542
Thread	u++-4.7	C++	105	296
	Presto-1.0	C++	56	108
	uSystem-4.4.3	C	90	180
Sim.	Awesime-2.0	C++	171	300
	C++SIM-1.7.2	C++	69	162
	CNCL-1.10	C++	179	369
3d	Crystal-0.10	C++	249	852
	Apprentice-0.5	C++	240	307
	VTK-2.1	C++	599	1074

Table 1: The libraries used in the experiments.

where R_i is the set of classes reachable from D_i in G (but excluding D_i).

Function matching is done similarly except that the graph G has an edge for every call from one function to another.

6 EXPERIMENTAL RESULTS

We have performed experiments to determine how well the name matching and similarity matching techniques work in practice. Specifically, we have compared libraries in four domains: graphics user interface (GUI), thread, simulation, and 3d graphics. In each domain, we have done pairwise comparisons among three libraries (although in general, our approach can compare more than two libraries at a time). Table 1 shows each library, its domain, language, as well as its class and function family counts.

Name Matching
Tables 2 and 3 show the results for name matching with classes and functions, respectively. In each table, we present a mix of quantitative and qualitative information: for each pairwise comparison between two libraries, we show the number of name matches as well as a list of some of these matches.

From the two tables, it is clear that name matching is good at identifying fundamental classes and functions in the libraries being compared. For example, in the GUI domain, name matching identifies classes Widget, Button, Menu, Window, Dialog and functions Hide, Show, Clip, Drag, Flush, Focus, Paint, Update, Cut ,and Paste.

However, name matching can also miss some fundamental domain concepts. For example, in the thread domain, the most important concept is that of a "thread", or equivalently, "task". Yet this concept is named BaseTask in u++, Thread in Presto, and Task in uSystem, so there is no corresponding name match in any of the comparisons. As we shall see, similarity matching can help in this regard.

Similarity Matching
Tables 4 and 5 show the results for similarity matching with classes and functions, respectively. In each table, we present

Domain	Libraries	# Matches	Class Name Matches
GUI	Qt, JX	20	Button, Cursor, Display, Image, Menu, Painter, Region, Widget, Window, …
	Qt, Kaffe	25	Button, Cursor, Dialog, Font, Event, Menu, PopupMenu, Scrollbar, Window, …
	JX, Kaffe	10	Button, CheckBox, Container, Cursor, Image, Menu, ScrollBar, Window, …
Thread	u++, Presto	4	Caddr, Condition, Lock, SpinLock
	u++, uSystem	7	Cluster, Condition, Event, Lock, Message, Processor, Semaphore
	Presto, uSystem	5	Condition, Lock, Monitor, Stack, Stderr
Sim.	Awesime, C++SIM	4	Histogram, Quantile, Semaphore, Thread
	Awesime, CNCL	13	Binomial, Event, Geometric, Histogram, Normal, Poisson, Random, Server, …
	C++SIM, CNCL	4	HashIterator, Histogram, Job, Queue
3d	Crystal, Apprentice	15	Base, Camera, Color, Image, Light, Line, Matrix, Plane, Texture, Vector, …
	Crystal, VTK	21	Camera, Component, Image, Light, Line, Matrix, Plane, Polygon, Texture, …
	Apprentice, VTK	20	Camera, Cylinder, Image, Light, Line, Material, Matrix, Plane, Texture, Transform, …

Table 2: Experimental results for class name matching.

Domain	Libraries	# Matches	Function Name Matches
GUI	Qt, JX	130	Activate, Cut, Display, Drag, Flush, Focus, Hide, Paste, Raise, Show, Update, …
	Qt, Kaffe	113	Activate, Align, Clip, Flush, Hide, Paint, ProcessEvent, Show, Update, …
	JX, Kaffe	76	Accept, Activate, Filter, Flush, FontList, Hide, Move, Show, Update, …
Thread	u++, Presto	17	Clock, Fork, Fp, Lock, MemoryAlign, PageSize, Pc, Pid, Sleep, Stack, Time, …
	u++, uSystem	43	Acquire, Block, Delay, Fp, Idle, Message, Migrate, P, Pc, Pid, Stack, Yield, …
	Presto, uSystem	12	Flags, Fp, Monitor, Pc, Pid, Ready, Resume, Stack, StackSize, Time, Wait
Sim.	Awesime, C++SIM	28	ArrivalTime, Await, Confidence, Lock, Release, ServiceTime, Signal, Trigger, …
	Awesime, CNCL	38	Confidence, Delta, LogMean, LogVariance, Priority, Seed, Time, Variance, …
	C++SIM, CNCL	19	Buffer, Confidence, Resize, StartTime, Sum, Sync, Time, Uniform, Variance, …
3d	Crystal, Apprentice	48	Intersect, Inverse, Normalize, Perspective, Show, Transform, Translate, …
	Crystal, VTK	94	Draw, Flush, Inverse, Normalize, Shift, Sync, Transform, Translate, …
	Apprentice, VTK	69	BackBuffer, Cross, Dot, Invert, Normalize, Rotate, Scale, Transform, …

Table 3: Experimental results for function name matching.

several similarity matches that, in our judgment, are particularly "informative" or "illuminating". By this, we mean that the two components in question either (1) serve essentially the same purpose in each library or (2) at least share some important role(s). We exclude matches with components of the same name; this case is already handled by name matching.

Each similarity match is written as "$A/B\ (n)$" where A is the component in the first library, B is the component in the second library, and n is the rank of the match according to the similarity measure defined in Section 5. We only show similarity matches that rank among the top 25.

First, observe those matches that indicate classes that serve a similar purpose. For example, note that the important domain concept thread/task is identified by the following similarity matches: "BaseTask/Thread (1)" for u++ and Presto, "BaseTask/Task (11)" for u++ and uSystem, and "Thread/Task (4)" for Presto and uSystem. Other notable matches include "MultiLineEdit/TextEditor (3)" for Qt and JX, "Condition/Semaphore (2)" for Awesime and C++SIM, and "Material/MatProp (4)" for Apprentice and VTK.

Second, observe those matches that indicate classes with (only) shared role(s). For example, the match "MachContext/Thread (2)" for u++ and Presto actually represents that role which manages a separate machine context for each thread (which includes a program counter, separate stack, etc.). Other role-based matches include "Widget/Container (8)" for Qt and JX, "SimpleStatistic/Variance (10)" for Awesime and C++SIM, and "Dview/Camera (10)" for Crystal and VTK.

Function similarity matching also yields useful matches such as "PaintEvent/Draw (1)" for Qt and JX, "PaintEvent/Paint (2)" for Qt and Kaffe, and "Draw/Paint (7)" for JX and Kaffe, all of which denote the corresponding member function in each library that one overrides to render new widget types. Other notable matches include "W/Wait (2)" for u++ and Presto, "Variance/StandardDeviation (2)" for Awesime and C++SIM, and "Draw/Render (2)" for Crystal and VTK. As functions are more finely-grained than classes, most of these matches indicate functions that perform a similar function, although a few matches, such as "S/Broadcast (4)" for u++ and Presto, indicate (only) a shared role (which in this case is "signaling" a condition variable in the thread domain).

7 TOOL

Our approach is supported by CodeWeb, a tool we have built for assessing C, C++, and Java libraries. Given a set of two or more libraries, the tool automatically performs the name and similarity matching described in Section 4 and 5, respectively. To illustrate the use of CodeWeb in assessing libraries,

Domain	Libraries	Class Similarity Matches
GUI	Qt, JX	Widget/Window (1), Printer/JpsPrinter (2), MultiLineEdit/TextEditor (3), Widget/Container (8)
	Qt, Kaffe	Widget/Component (1), Color/IndexColorModel (3), Painter/Graphics (20)
	JX, Kaffe	Container/Component (2), Window/Component (7), Widget/Component (11), Window/Frame (13)
Thread	u++, Presto	BaseTask/Thread (1), MachContext/Thread (2), Processor/Process (3), MachContext/Stack (4)
	u++, uSystem	MachContext/Stack (1), MachContext/Task (6), BaseTask/Task (11)
	Presto, uSystem	Thread/Stack (1), Thread/Task (4), Process/Processor (5)
Sim.	Awesime, C++SIM	Condition/Semaphore (2), SimMux/Process (4), SimpleStatistic/Variance (10)
	Awesime, CNCL	DiscreteUniform/DiscUniform (4), BatchStatistic/BatchMeans (11), SimpleStatistic/Confidence (12)
	C++SIM, CNCL	Variance/Confidence (2), RandomStream/Gen (14), Variance/Statistics (18), Variance/Normal (18)
3d	Crystal, Apprentice	PolyPlane/Plane (4), Timer/ElapsedTime (13), CLights/Light (21), Dview/Camera (24)
	Crystal, VTK	Matrix/Transform (1), Dview/Camera (10), RgBPixel/Colour (18), RgBcolor/Colour (18)
	Apprentice, VTK	Matrix/Transform (1), Vecf/Math (2), Material/MatProp (4), MaterialIndex/MatProp (5)

Table 4: Experimental results for class similarity matching. The rank of each match is shown in paranetheses.

Domain	Libraries	Function Similarity Matches
GUI	Qt, JX	PaintEvent/Draw (1)
	Qt, Kaffe	PaintEvent/Paint (2), PaintEvent/Repaint (10)
	JX, Kaffe	Draw/Paint (7)
Thread	u++, Presto	W/Wait (2), SBlock/Broadcast (4), S/Broadcast (4), Acquire/Lock (8), S/Signal (14)
	u++, uSystem	
	Presto, uSystem	StBase/Base (8)
Simulation	Awesime, C++SIM	Variance/StandardDeviation (2)
	Awesime, CNCL	Beta/CnBeta (5)
	C++SIM, CNCL	StandardDeviation/Variance (3), Confidence/RelativeVariance (5), Gen/Uniform (19)
3d	Crystal, Apprentice	Draw/GlRender (1), Unit/Units (2), IntersectSphere/Intersect (3)
	Crystal, VTK	Draw/Render (2), Draw()/Update (5), Execute/Perform (23)
	Apprentice, VTK	GlRender/Render (2), GlRender/Draw (5), GlRender/Update (6)

Table 5: Experimental results for function similarity matching. The rank of each match is shown in parentheses.

we demonstrate the tool on two C++ GUI libraries, Qt 1.4 and JX 1.0.8. (These were among the libraries compared in Section 6.) Refer to Figure 1 for the remainder of this section.

CodeWeb uses the more precise name matching to generate *class* and *function views*, which not only show classes and functions but also relationships between them that are shared across libraries. Although not shown in the Figure 1, the results from the similarity matching are included as complementary information along with each view.

In both class and function views, CodeWeb represents classes in shaded rectangles while functions appear, with a "()" suffix, in unshaded rectangles. The association relationship between a class and a function is indicated by a black bidirectional arrow. Other relationships (such as inheritance and invocation) are represented by unidirectional arrows and can be direct or indirect; CodeWeb uses dark and light shading for direct and indirect relationships, respectively.

Class View

The class view, much like a class diagram, is primarily concerned with classes and relationships between. Specifically, given a set of two or more libraries, the class view contains all classes that match by name across the libraries and *only* those functions, if any, that are associated with these classes and that match by name in all the libraries.

The class view shows important functional concepts in the domain. In our example, the diagram includes such important classes as Widget, Button, MenuBar, ScrollBar, Image, Window, Painter, and Printer. We also see key functions that are associated with these classes. For example, we see that AdjustSize, Focus, Move, and Scroll are associated with Widget.

Moreover, we see fundamental relationships between classes such as the fact that Button and ScrollBar directly inherit from Widget (as indicated by the black double-edged arrows), while CheckBox, MenuBar, RadioButton, and Slider also inherit from Widget but in an indirect manner (as indicated by the gray double-edged arrows). Other relationships also shown include direct composition, with Widget containing Rect and Painter containing Point, and an indirect reuse relationship, with Window reusing Rect.

Function View

The function view, much like a call graph, is primarily concerned with functions and invocation relationships between them. The function view includes all functions that match by name across a set of libraries and *only* those classes, if any, that are associated with these functions and that match by name in all the libraries.

The function view is useful for identifying the roles being played by classes in the libraries — even if those roles are

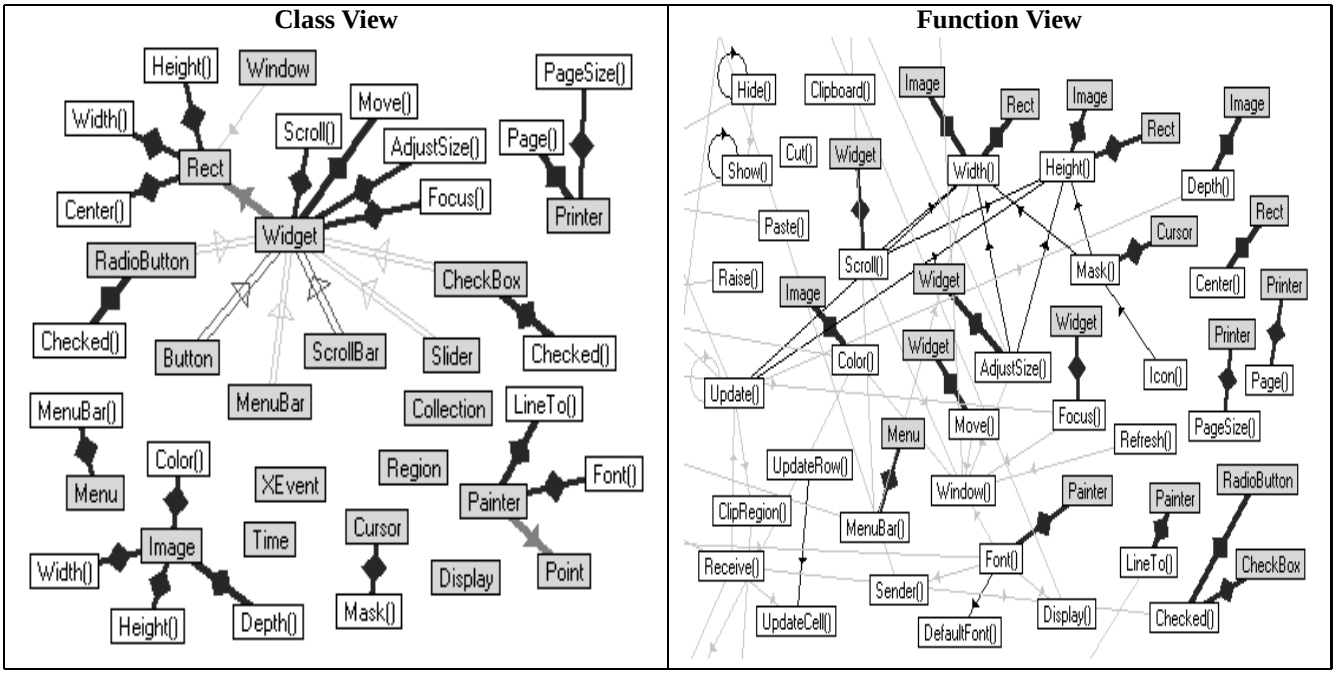

Figure 1: Class and function views for the Qt and JX libraries.

played by different classes. In our example, it is clear from the class view that Widget plays the "adjust size", "focus", "move", and "scroll" roles (as indicated by functions of the same name). However, it is only in the function view where we find roles such as "hide", "show", and "raise". In Qt, all three roles are played by Widget, whereas in JX, "hide" and "show" are played by Container while "raise" is played by Window. As another example, the roles "cut" and "paste" are played by MultiLineEdit in Qt but by TextEditor in JX.

Also observe some of the function invocations present in both libraries. For example, Font directly calls DefaultFont and indirectly calls Display, Sender, and Window. Functions with self-loops, such as Show and Hide, usually indicate code that defers a request to a parent class or that delegates a call to one of the instance variables. (Of course, self-loops may also indicate recursion.)

Links to the Source

We view the links to the source as a critical part of the system. Indeed, one can view the component and relationship matching as providing a starting point for exploration of the source code in the collection of libraries under consideration.

By clicking on any component in the class or function views, one can browse the members of its family in each library and access the corresponding source code fragment(s). And as mentioned earlier, we also present the results of similarity matching along with each view. The user can also click on similarity matches to see the corresponding family members and source.

For example, by clicking on the Painter class, one can easily compare the drawing primitives (which are members of Painter) that are supported by each library. One might determine, for instance, that while both Qt and JX support lines, rectangles, ellipses, and arcs, only Qt supports Bezier curves.

As another example, by clicking on the MultiLineEdit and TextEditor similarity match (shown in Table 4), the user can compare and contrast the editing capabilities of the MultiLineEdit class family (which is found in Qt and has one member QMultiLineEdit) and the TextEditor class family (which is found in JX and has two members, JTextEdtor and JXTextEditor).

Exploration of the source can also be useful in identifying "non-functional properties" such as extensibility, adaptability, modularity, flexibility, understandability, maintainability, etc. For example, we may want to know how easy it is to create new widgets in each library. One can do this by simply inspecting the source code of classes Button and ScrollBar which directly inherit from Widget in both Qt and JX.

Moreover, if one clicks on the Button class to explore its source in each library, one would notice that both libraries use implicit invocation to provide loose coupling between objects. It is well known that this improves adaptability, understandability, and maintainability. Further investigation would show that Qt allows one to connect a member that "emits" a signal to other members that will receive it, while JX only allows one to connect objects rather than object members.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have described how tool support can be used

to help developers directly compare and contrast libraries. A key part of this approach involves matching of similar classes and functions across libraries, as well as various relationships between them.

We have presented two matching techniques: name matching and similarity matching. While the name matching technique is more precise (and is therefore used to match relationships), the similarity matching is also important in identifying similar components with entirely different names that would not be found otherwise. We have performed comparisons on libraries in four domains and have found that name and similarity matching yield useful and complementary information.

Moreover, we have demonstrated our tool, CodeWeb, to show how our approach can be applied in practice. In particular, we have discussed the class and function views and the linking of components to the corresponding source in each library. We have also shown how one might assess two GUI libraries, Qt and JX, using our tool. For future research, we plan to conduct extensive user testing to see how useful our approach is in practice.

It is possible to match components and relationships across software systems in completely different ways than those described in this paper. Moreover, this can be done for purposes other than assessing a collection of libraries. For example, in [12], we describe a way to help developers reuse a *particular* software library by identifying components and relationships that are relevant across *several* user-selected example applications. It would be of interest to consider other ways to match components across different software systems for reuse related activities.

ACKNOWLEDGMENTS

We would like to thank Will Tracz and Michael Ernst for valuable feedback on this research.

REFERENCES

[1] T. J. Biggerstaff. Design Recovery for Maintenance and Reuse. *Computer*, 22(7):36–49, 1989.

[2] P. Chen, R. Hennicker, and M. Jarke. On the Retrieval of Reusable Software Components. In *2nd International Workshop on Software Reusability)*, pages 99–108. IEEE, 1993.

[3] W. B. Croft and D. J. Harper. Using Probabilistic Models of Document Retrieval Without Relevance Information. *Documentation*, 35(4):285–295, 1979.

[4] W. B. Frakes and R. S. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

[5] W. B. Frakes and B. A. Nejmeh. Software Reuse through Information Retrieval. In *20th Hawaii International Conference on System Sciences*, pages 530–535. IEEE, 1987.

[6] W. B. Frakes and T. Pole. An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, 20(8):617–630, 1994.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] R. Helm and Y. S. Maarek. Integrating Information Retrieval and Domain Specific Approaches for Browsing and Retrieval in Object-Oriented Class Libraries. In *OOPSLA*, pages 47–61, 1991.

[9] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *16th International Conference on Software Engineering*, pages 81–90. IEEE, 1994.

[10] R. Kazman and S. J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *5th International Conference on Software Reuse*. IEEE, 1998.

[11] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[12] A. Michail and D. Notkin. Illustrating Object-Oriented Library Reuse by Example: A Tool-Based Approach. In *13th IEEE International Conference on Automated Software Engineering*, 1998.

[13] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, 1995.

[14] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, 1987.

[15] G. Salton and C. S. Yang. On the Specification of Term Values in Automatic Indexing. *Documentation*, 29(4):351–372, 1973.

[16] S. Sparks, K. Benner, and C. Faris. Managing Object-Oriented Framework Reuse. *Computer*, 29(9):52–61, 1996.

[17] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

[18] A. S. Yeh, D. R. Harris, and M. P. Chase. Manipulating Recovered Software Architecture Views. In *Proceedings of the International Conference on Software Engineering*, pages 184–194, 1997.