Illustrating Object-Oriented Library Reuse by Example
A Tool-Based Approach

Technical Report UW-CSE-98-05-01

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

Abstract

Once an object-oriented library has been selected for a project, there is still the substantial problem of training developers to use it. In particular, reusing library classes often requires understanding (at least part) of the library design. For example, inheriting from a library class often requires knowing how that class interacts with other classes and what methods should be overridden and how.

In this paper, we present a tool-based approach that examines how example programs reuse a particular library. In particular, we (1) extract a unified class diagram for each example that merges the example class diagram with the library class diagram; (2) identify the reuse boundary in each unified class diagram, which shows instances of direct reuse of the library in the example; and (3) intersect each subset of the reuse boundaries to capture similarities and differences among library reuse patterns in the examples.

By browsing the reuse boundary intersections, developers can immediately see what aspects of the library are demonstrated by the examples. In particular, our approach can facilitate reuse by: (1) guiding the developer towards important library classes of general utility; (2) guiding the developer towards library classes particularly useful for a specific application domain; and (3) providing access to the relevant source code in each example for further inspection. All aspects of our approach are supported by CodeWeb, a reuse tool we have built for C++ and Java libraries.

1 Introduction

Once an object-oriented library[1] has been selected for a project, there is still the substantial problem of training developers to use it. Unlike function libraries, object-oriented libraries often require some understanding of their design. This is particularly true with frameworks as they tend to impose a structure on the application. In particular, reusing an object-oriented library often requires knowing how library classes interact with each other what methods should be overridden and how.

Current methods for learning to reuse an object-oriented library include reading the manual and/or books as well as taking a course. Sparks, Benner and Faris give this advice for framework reuse:

> "...expect to train every staff member who will use a framework. This often means having individuals attend a one-week course at the vendor site or training large groups at the project site." [9, p. 54]

A crucial aspect of these techniques is that example programs are used throughout to illustrate how to reuse the library. Indeed, most libraries come with many example programs to get the developer started.

Moreover, the selection of the examples is important. Sparks, Benner and Faris say the following about demonstration code:

> "Demonstration code that is provided by the framework builder...tends to demon-

[1] To simplify the presentation, we use the term *library* to mean any large software component, including libraries, frameworks, applications, etc.

> strate the uses of all the features of the framework, rather than concentrating on how the framework is to be used in an application context." [9, p. 60]

In this paper, we present a tool-based approach that examines how user-selected example programs reuse a particular library. An important aspect of our approach is that it requires no extra effort on the part of the library developer and can be used with any existing object-oriented library. Our method is not intended to replace current techniques for training developers to reuse a library but, rather, to complement them.

Specifically, we (1) extract a unified class diagram for each example that merges the example class diagram with the library class diagram; (2) identify the reuse boundary in each unified class diagram, which shows instances of direct reuse of the library in the example (where the "boundary" is itself a class diagram); and (3) intersect each subset of the reuse boundaries to capture similarities and differences among library reuse patterns in the examples.

By browsing the reuse boundary intersections, developers can immediately see what aspects of the library are demonstrated by the examples. In particular, our approach can facilitate reuse by: (1) guiding the developer towards important library classes of general utility; (2) guiding the developer towards library classes particularly useful for a specific application domain; and (3) providing access to the relevant source code in each example for further inspection. All aspects of our approach are supported by CodeWeb, a reuse tool we have built for C++ and Java libraries.

The remainder of the paper is organized as follows. Section 2 presents the computation of unified class diagrams. Section 3 shows how to identify the reuse boundary in a unified class diagram. Section 4 describes how to intersect reuse boundaries. Section 5 discusses the tool, CodeWeb, and shows how it can be used to help reuse object-oriented libraries by example. Section 6 discusses related work. Section 7 summarizes the work, concluding with a number of open questions.

2 Extracting the Unified Class Diagram

The first step in our approach is to extract a *unified class diagram* for each example that merges the example class diagram with the library class diagram. In Section 3, we use this diagram to identify a "reuse boundary" between the example and library, which shows instances of direct reuse of the library in the example.

We use the terms "example" and "library" loosely; we only require that the example refers to some software layer that reuses another software layer, which we refer to as the library. For example, the "example" may actually be a layer in a framework that reuses another layer — the "library" — also in that same framework. As another example, we may be interested in seeing how an application reuses *several* toolkits, in which case the "example" is the application and the "library" is the collection of toolkits.

Since the example and library may themselves consist of several software systems, we need to avoid clashes in class names (which may occur when using namespaces in C++ and modules in Java). Consequently, we prefix each class name with the software system name in which it is defined. For example, a class Dialog from the application framework ET++ is referred to as ET++:Widget. Similarly, a class myDialog from an example ex1 is referred to as ex1'myDialog.

In what follows, we show how to construct the unified class diagram U_i of an example i and the library. Whenever we refer to the "example" in what follows, we mean example i. (Refer to Figure 1 for examples of unified class diagrams.)

The diagram U_i is a directed graph in which the nodes denote classes and member functions while the edges denote direct/indirect inheritance, reference, and membership relationships between them.

Classes: $a \in U_i$ iff a appears in the example/library.

Direct Inheritance: $a \Rightarrow b \in U_i$ iff classes a and b appear in the example/library and a directly inherits from b.

We don't distinguish between composition (i.e., aggregation) and reference (i.e., acquaintance). In general, it is not always possible to determine whether a reference is a composition.

Direct Reference: $a \mapsto b \in U_i$ iff classes a and b appear in the example/library and a directly references b (i.e., a has a member variable of type b or pointer to b).

A member function f cannot appear in isolation but must be associated with some class a, which is done

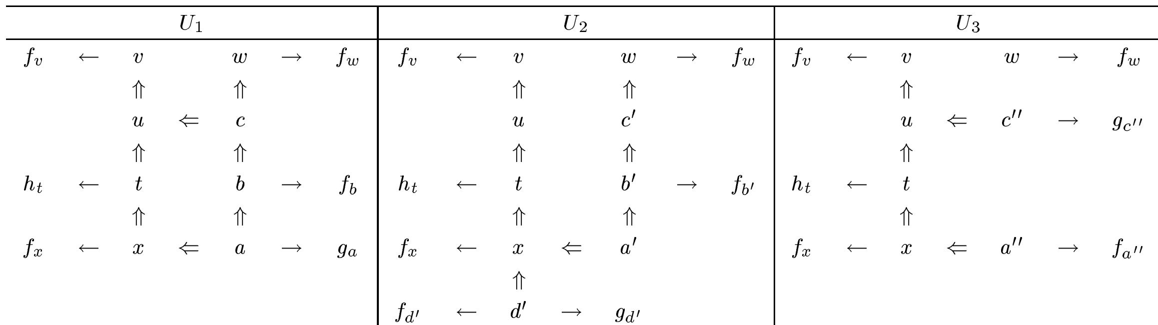

Figure 1: The unified class diagrams for three examples that reuse a library with classes t, u, v, w, and x. (We do not show redundant closure edges.)

by using the notation "f_a". The subscript a on f distinguishes different definitions/declarations of f in U_i. (See Figure 1.)

Member Functions: $f_a \in U_i$ iff class a appears in the example/library and a defines or declares a member function f.

Direct Membership: $a \rightarrow f_a \in U_i$ iff $f_a \in U_i$.

As discussed earlier, we want to intersect the reuse boundaries to capture similarities and differences among the library reuse patterns. To increase the likelihood that some relationships match across reuse boundaries, we consider not only *direct* relationships but also the *closure* of these relationships. For this reason, we add closure edges for inheritance, reference, and membership paths in the unified class diagram U_i as follows.

Closure Inheritance: $a \Rightarrow^+ b$ appears in U_i iff there is a path of length at least one from a to b over inheritance edges ($\Rightarrow$) in U_i.

Closure Reference: $a \mapsto^+ b$ appears in U_i iff there is a path from a to b over inheritance ($\Rightarrow$) and reference edges ($\mapsto$), that includes at least one reference edge, in U_i.

Closure Membership: $a \rightarrow^+ f_b \in U_i$ iff $b \rightarrow f_b \in U_i$ and either $a \Rightarrow^+ b \in U_i$ or $a = b$.

3 Identifying the Reuse Boundary

In Section 2, we have defined the unified class diagram U_i which merges the class diagrams of example i and the library. In this section, we show how to identify the *reuse boundary* in each unified class diagram, which shows instances of direct reuse of the library in example i.

The reuse boundary is itself a kind of class diagram. However, since we are not concerned about reuse within the example or library, the reuse boundary tends to be much smaller than the unified class diagram. In this way, the library reuse in the example is more apparent.

Although the reuse boundary for each example can be useful on its own, in Section 4 we intersect each subset of the reuse boundaries to capture similarities and differences among the library reuse patterns in the examples.

The reuse boundary is a function of three things: (1) the unified class diagram U_i; (2) a set of example classes E_i; and (3) a set of library classes L. (Observe $E_i \cap L = \emptyset$.) The reuse boundary, which we denote by $B_i(E_i, L)$, shows instances of direct reuse of the library classes L in the example classes E_i. The sets E_i and L need not include all the classes in example i and the library, respectively — specifying a subset of these classes allows one to narrow the results to only items of interest (which we do in Section 5).

In the remainder of this section, refer to Figure 2 which shows the reuse boundaries corresponding to the unified class diagrams in Figure 1.

3.1 Inheritance and Composition Reuse

The two most common techniques for reuse in object-oriented systems are class inheritance and composition [4, p. 18]. These are the primary kinds of reuse

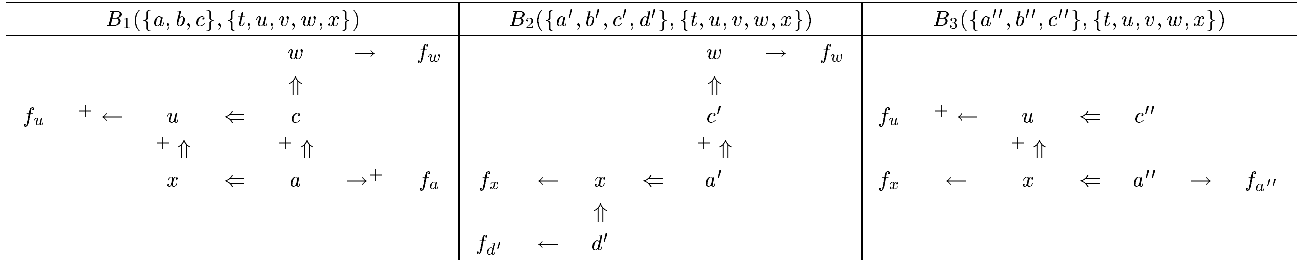

Figure 2: The reuse boundaries corresponding to the unified class diagrams in Figure 1. (We do not show redundant closure edges.)

represented by the reuse boundary.

We say that a class a *directly reuses* another class b if and only if a directly inherits from b or a directly references b. (As it is not always possible to tell whether a reference is a composition, we consider all references as instances of reuse.)

A class c appears in the reuse boundary if and only if it is involved in a direct reuse relation across the example/library boundary. (See Figure 2.) That is,

Classes: $c \in B_i(E_i, L)$ iff there exists $e \in E_i$ and $l \in L$ such that

- $c = e$ or $c = l$; and
- $e \Rightarrow l \in U_i$ or $e \mapsto l \in U_i$.

We include inheritance and reference relationships among classes in the reuse boundary. This may include instances of reuse within the example or within the library. However, such reuse does not determine which classes go into the reuse boundary; we only show such relationships among classes already in the boundary according to the definition above.

Direct Inheritance: $c_1 \Rightarrow c_2 \in B_i(E_i, L)$ iff $c_1 \in B_i(E_i, L)$, $c_2 \in B_i(E_i, L)$, and $c_1 \Rightarrow c_2 \in U_i$.

Direct Reference: $c_1 \mapsto c_2 \in B_i(E_i, L)$ iff $c_1 \in B_i(E_i, L)$, $c_2 \in B_i(E_i, L)$, and $c_1 \mapsto c_2 \in U_i$.

Closure Inheritance: $c_1 \Rightarrow^+ c_2 \in B_i(E_i, L)$ iff $c_1 \in B_i(E_i, L)$, $c_2 \in B_i(E_i, L)$, and $c_1 \Rightarrow^+ c_2 \in U_i$.

Closure Reference: $c_1 \mapsto^+ c_2 \in B_i(E_i, L)$ iff $c_1 \in B_i(E_i, L)$, $c_2 \in B_i(E_i, L)$, and $c_1 \mapsto^+ c_2 \in U_i$.

3.2 Member Function Overriding

Member function overriding is often used to customize the behavior of base classes in various ways and is a key part of inheritance reuse. However, it is often more difficult to override a member function with appropriate code than it is to simply call a function in a library class. For this reason, we show only overridden member functions in the reuse boundary (and not just any member function). In this way, a developer can look at member functions in the reuse boundary and immediately see which functions are commonly overridden in example classes and how (by looking at the corresponding source).

Rather than including all member functions in the reuse boundary that can potentially be overridden (as indicated by the "virtual" modifier in C++ or the lack of a "final" modifier in Java), we only include those member functions that are defined in the library and actually overridden in the example. (Observe that we omit member functions g and h in Figure 2 for this reason.) By showing only these member functions: (1) we show only overriding that spans the boundary between the example and library; (2) we show only those member functions which are typically overridden in practice; (3) we always have access to the source to see how such overriding is actually done; and (4) we reduce the number of member functions shown in the reuse boundary (thus reducing diagram clutter).

Before we introduce the rule for associating members with classes, we first define a function F_i that helps us determine member functions defined in the library and overridden in the example. In particular, for a library class $l \in L$, $F_i(l, E_i)$ yields the set of all member functions defined in an example class $e \in E_i$, or any descendent of some $e \in E_i$, that is a descendent of l in U_i. (In Figure 1, $F_1(x, \{a, b, c\}) = \{g\}$

and $F_2(x, \{a', b', c', d'\}) = \{f, g\}$.) Conversely, for an example class $e \in E_i$, $F_i(e, L)$ yields the set of all member functions defined in a library class $l \in L$, or any ancestor of some $l \in L$, that is an ancestor of e in U_i. (In Figure 1, $F_1(a, \{t, u, v, w, x\}) = F_2(a', \{t, u, v, w, x\}) = \{f, h\}$.)

Our rule for determining whether a member function f appears in the reuse boundary or not is complicated by several factors: (1) we want to associate each member with a class (e.g., f_a) to distinguish different declarations/definitions of f; (2) we want to take into account inheritance of members; and (3) we want to preserve member overriding relationships (but not introduce new ones that don't exist in the unified class diagram). Formally, the rule for member function associations is as follows.

Member Functions: $f_c \in B_i(E_i, L)$ iff

1. $c \in B_i(E_i, L)$;
2. there exists s such that
   - $c \rightarrow^+ f_s \in U_i$; and
   - for any ancestor d of c in $B_i(E_i, L)$ and any t such that $d \rightarrow^+ f_t \in B_i(E_i, L)$, we have $s \Rightarrow^+ t \in U_i$ (i.e., f_s overrides f_t in U_i)
3. and the following holds:
   - if $c \in L$ then $f \in F_i(c, E_i)$; and
   - if $c \in E_i$ then $f \in F_i(c, L)$.

Part (1) of the above definition ensures that we associate f with c only if c is a class in the reuse boundary. Part (2) ensures two things: that c inherits/defines f; and that associating f with c does not introduce an overriding relationship in the reuse boundary that implies an overriding relationship that doesn't exist in the unified class diagram. This is done in the following way: we check that for each ancestor d of c in the reuse boundary, it must be the case that c inherits/defines some f_s which overrides any f_t inherited/defined by d in the reuse boundary $B_i(E_i, L)$, where the "overriding" must occur in the unified class diagram U_i. (The definition is recursive in ancestors of c since we need to know that $d \rightarrow^+ f_t \in B_i(E_i, L)$.) Finally, part (3) ensures that: if c is a library class in L, then some descendent example class must override f in U_i; and conversely, if c is an example class in E_i, then some ancestor library class must define/declare f in U_i. (Observe

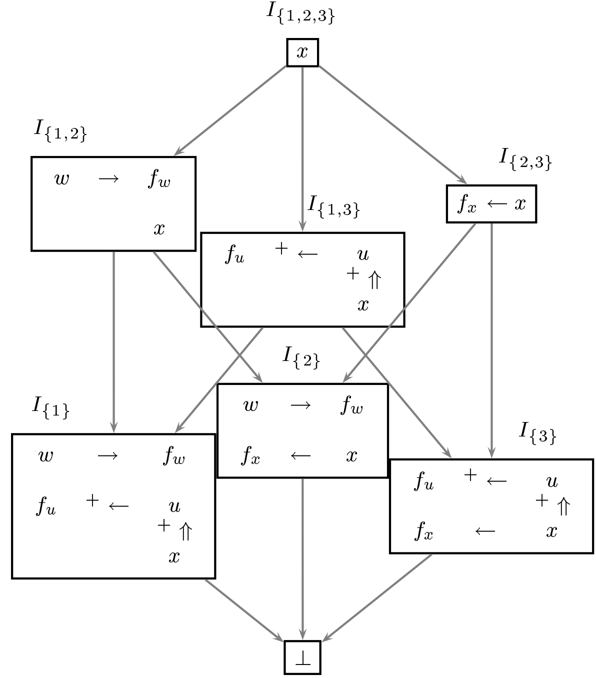

Figure 3: The lattice depicting the intersections of the reuse boundaries in Figure 2. (We do not show redundant closure edges.)

how in Figure 2 there is no f associated with x in $B_1(\{a, b, c\}, \{t, u, v, w, x\})$ since $f \notin F_1(x, \{a, b, c\})$. Also observe how we have a associated with f in $B_1(\{a, b, c\}, \{t, u, v, w, x\})$ but not a' associated with f in $B_2(\{a', b', c', d'\}, \{t, u, v, w, x\})$ since that would imply an overriding relationship with f_x which does not occur in U_2.)

The direct and closure membership relationships are defined as follows:

Direct Membership: $c \rightarrow f_c \in B_i(E_i, L)$ iff $f_c \in B_i(E_i, L)$ and $c \rightarrow f_c \in U_i$.

Closure Membership: $c \rightarrow^+ f_d \in B_i(E_i, L)$ iff $f_d \in B_i(E_i, L)$ and either $c \Rightarrow^+ d \in B_i(E_i, L)$ or $c = d$.

4 Intersecting Reuse Boundaries

In Section 3, we have shown how to identify the reuse boundary $B_i(E_i, L)$ in a unified class diagram U_i for each example i that reuses a particular library. In

this section, we show how to intersect each subset of the reuse boundaries to capture similarities and differences among the library reuse patterns in the examples.

We use the notation I_X to denote the intersection of the reuse boundaries $B_i(E_i, L)$ over $i \in X$. We compute I_X for every subset X of the set of examples, and we arrange the intersections in a lattice. A node I_X is an ancestor of I_Y in the lattice if and only if $X \supset Y$. (See Figure 3 which shows the lattice of intersections for the reuse boundaries in Figure 2.)

We only include library classes in I_X and these must be present in all the reuse boundaries involved in the intersection:

Classes: $c \in I_X$ iff $c \in L$ and $c \in B_i(E_i, L)$ for all $i \in X$.

The inheritance and reference relationships between library classes in I_X are defined as follows.

Direct Inheritance: $c_1 \Rightarrow c_2 \in I_X$ iff $c_1 \Rightarrow c_2 \in B_i(E_i, L)$ for all $i \in X$.

Direct Reference: $c_1 \mapsto c_2 \in I_X$ iff $c_1 \mapsto c_2 \in B_i(E_i, L)$ for all $i \in X$.

Closure Inheritance: $c_1 \Rightarrow^+ c_2 \in I_X$ iff $c_1 \Rightarrow^+ c_2 \in B_i(E_i, L)$ for all $i \in X$.

Closure Reference: $c_1 \mapsto^+ c_2 \in I_X$ iff $c_1 \mapsto^+ c_2 \in B_i(E_i, L)$ for all $i \in X$.

We associate members with classes in a similar way as that described in Section 3. However, the rule is simpler since we need only concern ourselves with library classes.

Member Functions: $f_c \in I_X$ iff

1. $c \in I_X$;
2. for all $i \in X$, there exists an s_i such that
   - $c \rightarrow^+ f_{s_i} \in B_i(E_i, L)$;
   - for any ancestor d of c in I_X and any t_i such that $d \rightarrow^+ f_{t_i} \in I_X$, we have $s_i \Rightarrow^+ t_i \in B_i(E_i, L)$ (i.e., f_{s_i} overrides f_{t_i} in $B_i(E_i, L)$);
3. for all $i \in X$, we have $f \in F_i(c, E_i)$.

(Observe in Figure 3 that any intersection involving example 1 does not associate f with x since $f \notin F_1(x, \{a, b, c\})$.)

And finally, the direct and closure membership relationships are as follows:

Direct Membership: $c \rightarrow f_c \in I_X$ iff $f_c \in I_X$ and $c \rightarrow f_c \in B_i(E_i, L)$ for all $i \in X$.

Closure Membership: $c \rightarrow^+ f_d \in I_X$ iff $f_d \in I_X$ and either $c \Rightarrow^+ d \in I_X$ or $c = d$.

5 Tool

All aspects of our approach are supported by CodeWeb, a reuse tool for C++ and Java libraries. Given a library and a set of examples, the tool automatically: extracts the unified class diagrams as described in Section 2; identifies the reuse boundary $B_i(E_i, L)$ in each unified class diagram as explained in Section 3 (where E_i and L are the sets of all classes in example i and the library, respectively); and intersects each subset of the reuse boundaries and arranges the results in a lattice as shown in Section 4. Now, each intersection I_X in the lattice contains only library classes. If the user clicks on such a library class $l \in I_X$, then CodeWeb shows the reuse boundary $B_i(E_i, \{l\})$ for *each* example $i \in X$. (In other words, the tool shows the part of each example that directly reuses, by inheritance or composition, library class l.)

To illustrate how CodeWeb might be used to facilitate library reuse, we demonstrate the tool on ET++ 3b4, a C++ GUI application framework which also provides basic data structures and object input/output. The example applications, included with the framework, are: Draw, Write, Debugger (a point-and-click interface for line-oriented debuggers), and ProgEnv (the ET++ programming environment). As explained earlier, the tool builds a lattice of reuse boundary intersections for each subset of the examples (which yields 16 nodes in this case).

CodeWeb shows exactly one node of the lattice on the screen at a time. (See Figures 4 and 5.) The tool represents classes in shaded rectangles while overridden function members appear, with a "()" suffix, in unshaded rectangles. Reference and inheritance relationships between classes are shown by narrow and wide edges, respectively. Direct and indirect relationships are indicated by dark and light shading, respectively.

5.1 Learning from Different Examples

Figure 4 shows the lattice node representing the reuse boundary intersection of Draw and Write. These examples are different in the sense that the Draw application is mostly graphical while the Write application

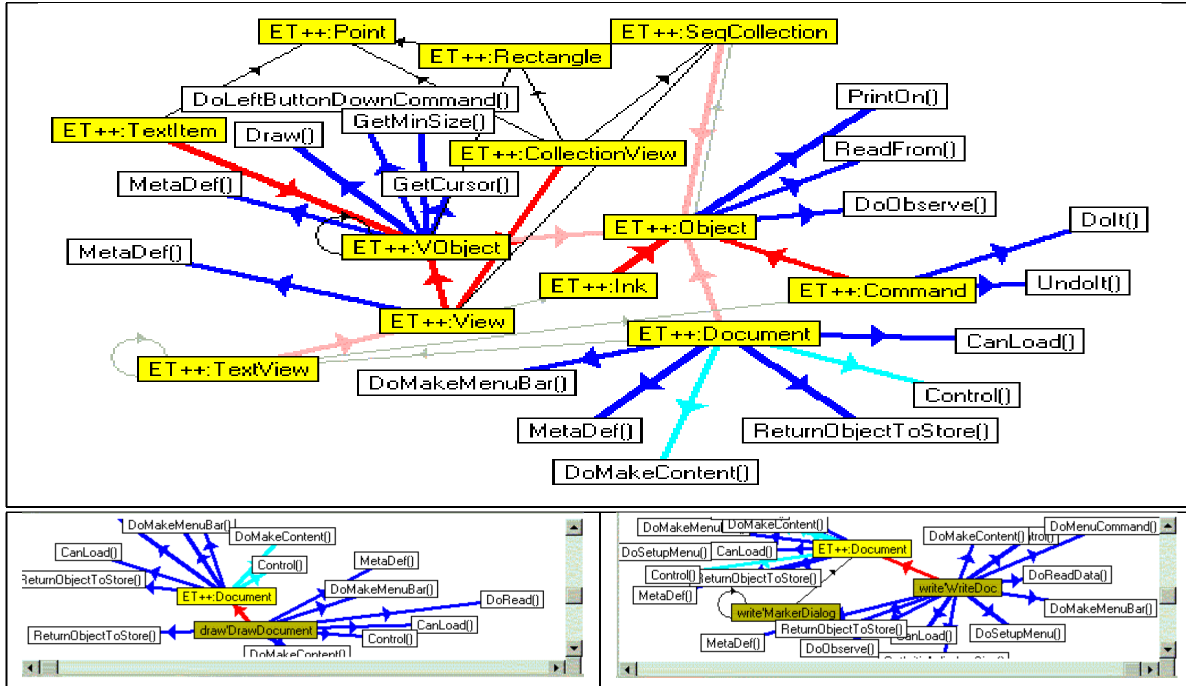

Figure 4: The lattice node depicting the reuse boundary intersection of Draw and Write is shown on top. The reuse boundaries depicting reuse of Document in the examples are shown on the bottom.

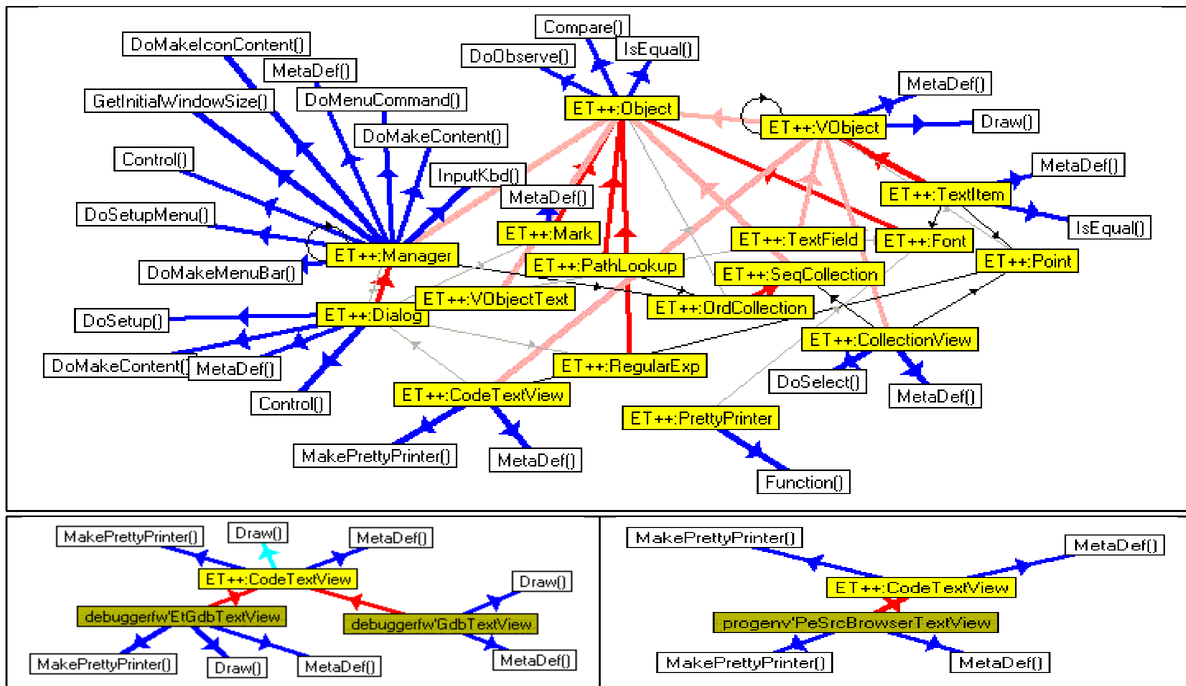

Figure 5: The lattice node depicting the reuse boundary intersection of Debugger and ProgEnv is shown on top. The reuse boundaries depicting reuse of CodeTextView in the examples are shown on the bottom.

is mostly concerned with text. Generally speaking, looking at the reuse boundary intersection of different application can be useful in determining important aspects of a library that are applicable to most applications, independent of their purpose.

For example, observe that the reuse boundary intersection of Draw and Write directs the user to important application framework classes such as Document, View, SeqCollection, CollectionView, Command, Object, VObject (i.e., visual object) as well as relationships between them such as the fact that CollectionView inherits from View and has a reference to SeqCollection. Also, observe the member functions shown which are overridden in both Draw and Write.

From such information, one can learn important aspects about writing applications using this framework. For example, it is apparent that the framework provides support for undo since both applications define commands by inheriting from Command and overriding the member functions DoIt() and UndoIt(). One can also infer that programming an application using ET++ involves separating its model (i.e., data structure) from its view (the way it is depicted on the screen). One can verify this by examining the source to Document and View, respectively. Indeed, further examination would show that a document may have several (possibly different) views and that this principle is used throughout the framework in more finely-grained ways, as with SeqCollection and CollectionView.

However, identifying and understanding the purpose of important ET++ classes is not enough. The user still needs to know how to write code that reuses such classes. To do this, one can click on a library class to see how it is reused in each example. (See Figure 4.) For example, by clicking on Document we see that both Draw and Write inherit from Document and override key member functions such as DoMakeMenuBar(), ReturnObjectToStore(), and Control(). By clicking on a class in the reuse boundary, one can see the corresponding source code. By doing this one would find that DoMakeMenuBar() creates the applications menu bar, ReturnObjectToStore() returns that aspect of the model state that is to be stored on disk, Control() handles user events, etc.

5.2 Learning from Similar Examples

Figure 5 shows the lattice node representing the reuse boundary intersection of Debugger and ProgEnv. These examples are similar in the sense that both Debugger and ProgEnv are C++ software development tools. Generally speaking, looking at the reuse boundary intersection of similar applications can be useful in determining aspects of a library that are useful for this class of applications (in addition to those of more general utility).

For example, observe that the reuse boundary intersection of Debugger and ProgEnv directs the user to classes relevant to the software development tool domain such as CodeTextView, RegularExp, and PrettyPrinter. (In addition, we also find important classes of more general utility such as Dialog, Font, TextField, and Manager.)

Again, by clicking on one of these classes, one can see how it is reused in each example. For example, by clicking on CodeTextView, we see that both Debugger and ProgEnv inherit from CodeTextView and override member functions such as MakePrettyPrinter() which determines which pretty printer to use (which is a subclass of the PrettyPrinter class mentioned above). Finally, observe that Debugger provides two examples of reuse of the CodeTextView class.

6 Related Work

We know of no tools that examine how examples reuse a library to give insight into how one might reuse that library in a new application. However, there has been much research into tools for software reuse, and we discuss some of them in what follows.

Perhaps the most well-known approaches are those concerned with finding a suitable component (such as a function or class) in a library that fits a particular need. Such work includes tools that use free-text indexing [3], facets [8], signature matching [11], and formal specifications [2]. While our approach is not query-based, browsing reuse boundary intersections for example programs would show important classes and relationships between them in the library — *even if the developer does not know exactly what kinds of classes to look for.*

Moreover, as discussed earlier, reusing an object-oriented library requires some understanding of its design, so tools that help a developer examine an individual library in terms of structure can be useful [1, 6, 7, 10]. However, object-oriented libraries are often huge, consisting of hundreds or thousands of classes with non-trivial relationships between them. The developer would rather just understand that part of the library that is crucial to the task at hand rather than examine the structure of the entire library.

In our approach, The lattice of reuse boundary intersections shows "views" of the library design that are relevant to various subsets of the example programs (and so the developer can see the library design at various levels of detail). Moreover, the developer can pick examples to show those aspects of the library that are of interest.

More recently, "exemplars" have been suggested as a way of library reuse [5]. An exemplar is an executable visual model consisting of one or more instances of at least one concrete class for each abstract class in a library. For example, a GUI library might have an exemplar that consists of a window object together with its menu bar, tool palette, canvas, and some widgets on the canvas. (Since libraries have only a few abstract classes, a small number of instances suffice in creating an exemplar.) The developer would not only explore the structural relationships in the exemplar but also the collaborative relationships among objects by observing message passing among these objects.

The problem with an exemplar is that it is too abstract and is not representative of any particular application reusing the library. Our approach can be used to see how real example applications reuse the library at various levels of abstraction (which may include many concrete classes).

7 Conclusions and Future Work

In this paper, we have described a tool-based approach that examines how example programs reuse a particular library. Our approach is not intended to replace current methods for training developers to reuse a library but, rather, to complement them.

We have demonstrated how a developer might use CodeWeb to help reuse the ET++ application framework. For future research, we plan to conduct extensive user testing to see how developers would use the tool in practice and how much utility it provides them in the reuse process.

It is possible to run CodeWeb on different versions of the same example to see how it reuses a library over time. One may expect that early versions of the example would reuse the most fundamental parts of the library (which tend to be of general utility) while later versions would reuse parts more specific to that application. It would be interesting to explore this idea further.

From our research, it appears that class diagrams are an appropriate high-level abstraction for comparing examples that reuse a library. However, it may also be interesting to consider collaboration diagrams, sequence diagrams, state diagrams, etc. (although these would be harder to extract automatically from the source).

One may extract class diagrams and intersect them in different ways than that described in this paper. Indeed, CodeWeb has a different mode in which it can help select one of several libraries to use in a particular application by intersecting the library class diagrams in a "relaxed way".

For future research, it would be interesting to provide yet another mode that helps developers determine if reusing a *set* of libraries S would result in problems (and to find workarounds if any). For example, one might run the tool on applications that reuse several libraries, at least one of which is in S.

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

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

[5] D. Gangopadhyay and S. Mitra. Understanding Frameworks by Exploration of Exemplars. In *7th International Workshop on Computer-Aided Software Engineering*, pages 90–99. IEEE, July 1995.

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

[7] 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.

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

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

[10] 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.

[11] A. M. Zaremski and J. M. Wing. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.