Data Mining Library Reuse Patterns in User-Selected Applications

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

Abstract

In this paper, we show how data mining can be used to discover library reuse patterns in user-selected applications. This can be helpful in building and debugging applications that use a particular library by observing how other developers have used that library in their applications.

Specifically, we consider the problem of discovering *association rules* that identify library components that are often reused in combination by application components. For example, such a rule might tell us that application classes that inherit from a particular library class often override certain member functions.

By querying and/or browsing such association rules, a developer can discover patterns for reusing library components. We illustrate the approach using our tool, CodeWeb, by demonstrating characteristic ways in which applications reuse components in the ET++ application framework.

Keywords

Software libraries, software reuse, data mining.

1 Introduction

Once a software library has been selected for a project, there is still the substantial problem of training developers to use it. This process is complicated by the fact that using a software library often requires some understanding of its fundamental components, their relationships, and various interactions between them. This is particularly true with frameworks which tend to impose a structure on the application. For example, one often needs to know the fundamental classes in a framework, their interactions, and which methods should be overridden and how.

Current methods for learning to use a software 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. [23, p. 54]

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

Such example programs — whether in manuals/books or packaged with libraries — are particularly helpful because they demonstrate characteristic reuse of the library components by experienced software developers. However, they also tend to be toy programs which limits their scope considerably. For example, a user that needs to develop an application that makes heavy use of text drawing primitives may not be satisfied with toy examples that demonstrate only the most rudimentary concepts of the GUI domain.

One can explore additional reuse experience by going beyond toy examples distributed with a library and actually finding real-life applications, written by others, that also use that library. With the emergence of the open source movement, such applications (and their source code) are now in abundance on the internet. This is also possible in large software companies where developers in one group may learn from applications written by another group.

However, unlike toy examples, it is more difficult to learn characteristic reuse patterns in larger applications since there is so much source to browse. To be sure, there is valuable reuse experience to be gained, but it is *implicitly* encoded and scattered throughout thousands of lines of code.

For this reason, it is desirable to have tools that tell us *explicitly* the characteristic library reuse patterns in large applications.

To this end, we have explored techniques used in *data mining* which are designed to find patterns in vast collections of data. The primary motivation for data mining has been in its potential to give a business a competitive advantage by better utilizing customer data [13, 15].

In this paper, we show how data mining can be used to discover library reuse patterns in user-selected applications. This can be helpful in building and debugging applications that use a particular library by observing how other developers have used that library in their applications.

Specifically, we consider the problem of discovering *association rules* that identify library components that are often reused in combination by application components. For example, such a rule might tell us that application classes that inherit from a particular library class often override certain member functions.

By querying and/or browsing such association rules, a developer can discover patterns for reusing library components. Moreover, the association rules can be used to automatically warn developers when their application reuses library components in a way that differs from characteristic usage in applications written by others.

The paper is organized as follows. Section 2 introduces the field of data mining and discusses the problem of mining association rules. Section 3 presents the concept of "reuse boundaries" which we developed in earlier work [17]. Section 4 shows how to mine association rules in reuse boundaries to discover characteristic library reuse patterns in existing applications. Section 5 presents experimental results. Section 6 discusses related work. Section 7 summarizes the work, concluding with a number of open questions.

2 Data Mining

Data mining may be defined as follows:

> The process of discovering meaningful new correlations, patterns, and trends by sifting through large amounts of data stored in repositories and by using pattern recognition technologies as well as statistical and mathematical techniques. [16]

Data mining is widely used in business to gain a competitive edge. For example, credit card companies use data mining to approve credit card applications, analyze credit holders' buying behavior, and detect fraud. As another example, retailers use data mining to understand customers' buying habits and preferences.

An effective data mining application in the retail environment is *shopping basket analysis*. Progress in bar-code technology has made it possible to store *basket data* that stores items purchased on a per-transaction basis. By using data mining technology, one can find patterns in items that are bought in combination.

Shopping basket analysis is often done by mining *association rules* [1, 2]. Formally, the problem is the following. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$.

An *association rule* is an implication of the form $\left(\bigwedge_{x \in X} x\right) \Rightarrow \left(\bigwedge_{y \in Y} y\right)$, which we write more compactly as $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$.

For example, suppose that people who purchase bread and butter also purchase milk. In that case, the corresponding association rule is "bread∧butter⇒milk". The antecedent of the rule X consists of bread and butter and the consequent Y consists of milk alone.

Such rules are useful for analyzing data. For example, to determine how one might boost the sales of milk, one could look for rules that have "milk" in the consequent. To determine the impact of discontinuing the sale of butter, one could find all rules that have "butter" in the antecedent.

(Incidently, observe that as we are using sets throughout, an item that occurs multiple times in a transaction is not treated any differently from one that occurs only once.)

We say that a rule $X \Rightarrow Y$ holds in the transaction set D with *confidence* $c\%$ if $c\%$ of transactions in D that contain X also contain Y. The rule $X \Rightarrow Y$ has *support* $s\%$ in the transaction set D if $s\%$ of transactions in D contain $X \cup Y$.

Returning to our example, suppose we find that in 90% of transactions in which customers purchase bread and butter, they also purchase milk. Moreover, say that 5% of transactions include all three items: bread, butter, and milk. In that case, the confidence of the rule "bread∧butter⇒milk" is 90% while its support is 5%.

Support should not be confused with confidence. While confidence is a measure of the rule's strength, support corresponds to its statistical significance [1, p. 208].

For example, a rule $a \wedge b \Rightarrow c$ may have much higher confidence than $a \Rightarrow c$, which means that whenever we encounter a in a transactions, it is more likely we find c if b is also present. So, in that sense $a \wedge b \Rightarrow c$ is stronger than $a \Rightarrow c$ and we should take it more seriously in our analysis of the data. (As a side note, this notion of rule strength is different from logical implication in which case $a \Rightarrow c$ would be considered stronger because it implies $a \wedge b \Rightarrow c$.)

Now, it may be that $a \Rightarrow c$ has much higher support than $a \wedge b \Rightarrow c$. This means that the confidence estimate for $a \Rightarrow c$ is more reliable than that for $a \wedge b \Rightarrow c$. So, while $a \wedge b \Rightarrow c$ may be a much stronger rule, we should still consider weaker rules, such as $a \Rightarrow c$, for which we are more certain of their confidence. In this sense, support is a measure of statistical significance. (However, it is by com-

paring support numbers across several rules that we obtain insight into the relative statistical significance of the rules; individual support numbers on their own are not as helpful.)

Given a set of transactions D, the problem of mining association rules is the following:

> Generate all association rules that have support and confidence greater then some user-specified minimum support and minimum confidence.

Several algorithms have been presented in the literature for finding all such association rules [1, 2, 3, 4, 24]. The particular algorithm used in our experiments is *Apriori* [2], which works in two phases: (1) it finds all sets of items that have support above the minimum support; and (2) it uses these sets to generate all rules whose confidence is above the minimum confidence. For additional details on this and other algorithms, see the references cited above.

In Section 4 we mine association rules to identify library components that are often reused in combination by application components. For example, such a rule might tell us that application classes that inherit from a particular library class often override certain member functions. Before showing how this is done, we first introduce the notion of "reuse boundaries" in the following section.

3 Reuse Boundaries

In this section, we present the concept of *reuse boundaries* which we developed in earlier work [17]. Reuse boundaries show reuse relationships that cross from one software system to another. That is, those reuse relationships that are at the "boundary" of the two software systems. We use a directed graph to denote a reuse boundary, where the nodes represent components and the edges represent reuse and membership relationships between them.

3.1 Boundary Nodes

Each node denotes a *component*, which is either a class or a function. We consider any type as a class, whether it appears in the source as a struct, class, interface, or union. Moreover, we include all functions, whether they are members of a class or not.

3.2 Boundary Edges

Edges represent reuse and membership relationships. By reuse relationships, we mean class inheritance and instantiation as well as function invocation and overriding. We elaborate on these relationships in what follows.

3.2.1 Membership

In object-oriented languages, classes contain member functions. If class C contains a member function f, then we also say that f is a member function of C; the relationship goes both ways.

3.2.2 Class Inheritance and Instantiation

The two most common techniques for reuse in software libraries are class inheritance and instantiation. While we use the familiar notion of "inheritance", we mean something quite specific by "instantiation".

Specifically, we say that class or function A *instantiates* class B if and only if (1) it declares a (possibly pointer) variable of type B; and (2) that variable denotes a new instance of B created by A. It is not sufficient to merely declare a pointer to an instance of B created by a class/function other than A. Finally, observe that if A is a class, then the instantiation relationship reduces to the composition relationship since B is an intrinsic part of A.

3.2.3 Function Invocation and Overriding

The invocation relationship is a reuse relationship that indicates a call from one function to another. We also look for the member function overriding relationship which acts like a callback from the base class to the derived class. For example, if class A inherits from base class B and overrides f, then we can view B as making a callback into class A. The reuse relationship is backward since B's f reuses A's f (since any calls to it are diverted to A's f).

3.3 Boundary Graph

Given two software systems (such as a software library and application), the *reuse boundary* between them contains: (1) components in the software systems that are involved in a direct reuse relationship that crosses from one software system to the other (in either direction); and (2) all relationships (including membership relationships) between components determined to be in the reuse boundary.

For example, an application may have a component myWidget that inherits from a library component Widget. In such a case, Widget and myWidget are in a direct reuse relationship that crosses from one software system to another, so both components go into the reuse boundary as well as the inheritance relationship between them (directed from myWidget to Widget).

The reuse relationship may also go the other way from the library to the application. This is the case with the overriding relationship. Suppose Widget defines paint() which is overridden in myWidget. In that case, there is

an overriding relationship from the paint() member function in Widget to the paint() member function in myWidget. Consequently, both paint() member functions go into the reuse boundary along with the overriding relationship between them (directed from Widget's paint() to myWidget's paint()).

Finally, since we have determined that Widget, myWidget, and their member functions paint() are in the reuse boundary, we also include the bidirectional membership relationship between Widget and its member function paint() as well as that between myWidget and its member function paint().

4 Mining Reuse Boundaries

Now that we have introduced association rule mining in Section 2 and reuse boundaries in Section 3, we can demonstrate how to mine association rules in reuse boundaries to discover library reuse patterns in existing applications. Specifically, we will identify library components that are often reused in combination by application components. We do this in a way analogous to that described in Section 2 for discovering items that are typically purchased together in basket data.

Suppose we have a library L and sample applications $A_1, \ldots, A_n$. We first compute reuse boundaries $B(L, A_1), B(L, A_2), \ldots, B(L, A_n)$. To simplify the exposition, suppose no two applications share an identical application component; that is, $a \in A_j$ implies $a \notin A_k$ for $k \neq j$.

Now, for every reuse boundary $B(L, A_j)$ and for each application component $a \in B(L, A_j)$, we construct a transaction $T_a = \{i_1, \ldots, i_m\}$ where the items are those library components involved in a direct reuse relationship with a in $B(L, A_j)$. Each item i_k is of the form <type of reuse relationship>:<library component involved in reuse relationship>.

For example, if an application component myWidget inherits from a library component Widget and overrides its member function paint(), then we construct a transaction T_{myWidget}={class_inherits:Widget, member_function_overrides:paint()}.

Now, given the set of transactions $D = \{T_a | a \in B(L, A_j)$ for some $j\}$, we generate all association rules that have support and confidence greater than some user-specified minimum support and minimum confidence.

Returning to our example, suppose we find that: (1) {class_inherits:Widget, member_function_overrides:paint()} occurs as a subset of sufficiently many transactions to satisfy the support requirement; and (2) sufficiently many transactions that contain class_inherits:Widget also contain member_function_overrides:paint() to satisfy the confidence requirement. In such a case, we would generate the association rule class_inherits:Widget $\Rightarrow$ member_function_overrides:paint().

Application	Lines	Transactions
Draw	5,157	114
Write	5,016	109
ProgEnv	5,721	177
DebuggerFW	10,939	211
FileBrowser	1,565	62
IconEdit	1,781	56
TroffTool	1,210	38
ER	698	33
BrowseFW	463	21
VObEdit	371	20
Total	32,921	841

Table 1. Sample applications used to discover reuse patterns in the ET++ framework.

5 Experimental Results

To illustrate how mining reuse boundaries can be helpful in finding library reuse patterns, we have performed experiments using the ET++ application framework [25]. This C++ framework provides not only GUI components but also ones for basic data structures and object input/output. The particular version that we used in our experiments is 3b4, which consists of 52,724 lines of code.

We used ten sample applications to discover characteristic reuse patterns in ET++: Draw, Write, ProgEnv, DebuggerFW, FileBrowser, IconEdit, TroffTool, ER, BrowseFW, and VObEdit. Although these are among the examples distributed with the framework, they are not toy programs. In particular, seven of these programs contain over 1,000 lines of code, four contain over 5,000 lines, and one has over 10,000 lines of code. The combined line count for all applications is 32,921. The number of transactions, as defined in Section 4, for all ten applications is 841. Refer to Table 1 for individual application statistics.

We have built a reuse tool, CodeWeb, which follows the methodology described in Section 4. We ran the tool on ET++ and the ten sample applications to generate association rules with confidence of at least 25% and support of at least .35%. (As there are 841 transactions, a support of .35% means rules must be supported by at least three transactions.) To keep the complexity of the computation down, we restricted the form of the rules to a single literal in the antecedent and a single literal in the consequent. The tool generated 8477 such rules.

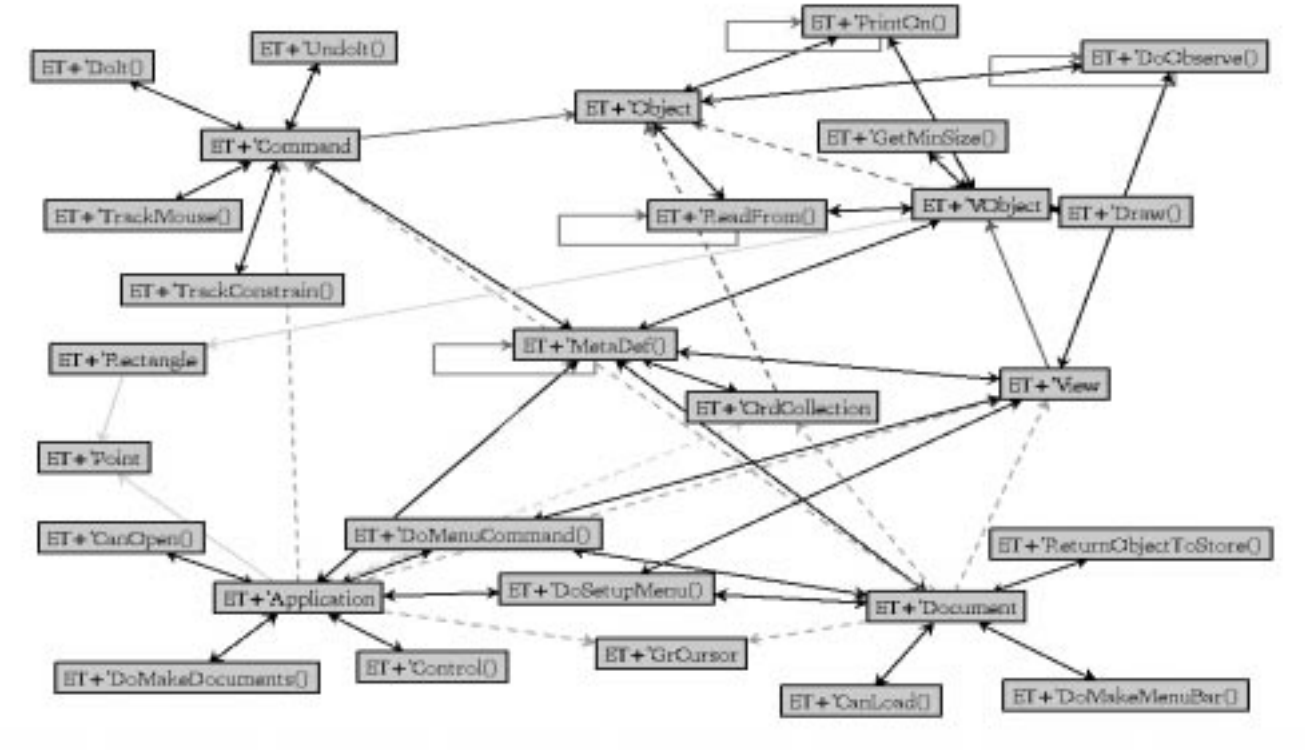

Figure 1. The reuse boundary intersection for Draw and Write.

Clearly, browsing thousands of association rules to identify reuse patterns is not feasible. One possibility is to use these rules to automatically check whether an application being developed reuses library components in a way that is consistent with characteristic usage in the sample applications. Moreover, to reduce false warnings, one might generate only association rules with sufficiently high confidence and support.

Another possibility is to manually explore the rules through a combination of querying and browsing. We follow this approach in the remainder of this section. (Of course, in doing so, we also provide evidence that an automatic checker has potential utility while building and debugging applications that use a particular library.)

In what follows, we take a two step approach to exploring reuse patterns:

1. we examine the reuse boundaries to identify fundamental library components; and

2. we use a combination of querying and browsing of the association rules to discover characteristic reuse patterns involving such components.

5.1 General Reuse Patterns

In this section, we consider reuse patterns of a general nature. In particular, we are interested in those reuse patterns that we expect to find in a wide variety of applications based on the ET++ application framework. One way to identify such patterns is to begin by finding fundamental library components that are reused in different kinds of applications.

To this end, we consider the reuse boundaries for sample applications Draw and Write. These applications are different in the sense that the Draw application is mostly graphical while the Write application is mostly concerned with text. Generally speaking, looking at different applications can be useful in determining important aspects of a library that are applicable to most applications, independent of their purpose.

While we can look at the reuse boundaries separately for Draw and Write, we shall first consider components in the intersection of the two boundaries. The reuse boundary intersection contains those library components (and their relationships) that are reused in both applications. Consequently, they tend to be particularly fundamental.

Figure 1 shows the reuse boundary intersection of Draw and Write. Graphically, our tool represents components in

shaded rectangles and distinguishes classes from functions by a "()" suffix appended to function names. Instantiation and inheritance relationships are shown by light and dark edges, respectively. Direct and indirect relationships are indicated by solid and dashed lines, respectively. Self-loops indicate method overriding in the library. Bidirectional arrows indicate membership relationships.

Observe that the reuse boundary intersection directs the user to important framework classes such as Document, View, Command, Application, Object, VObject (i.e., visual object) as well as relationships between them such as the fact that VObject instantiates Rectangle and inherits from Object.

The second step in our technique is to use a combination of querying and browsing to identify association rules pertaining to these fundamental library components. For example, if we were interested in reusing the Command class, we could perform a query that finds all association rules with "class_inherits:Command" in the antecedent and where the output is sorted in decreasing order by rule confidence. One might then browse the results and perform additional queries.

We have followed such a procedure to identify association rules for classes Command, Application, Document, and View. A sampling of the rules is shown in Table 2. (In addition to the support percentages, we also include the actual number of transactions supporting each rule in parentheses.) From these rules, one can learn important aspects about writing applications using ET++.

For example, while it may be apparent from Figure 1 that ET++ provides support for undo/redo (since class Command has members DoIt() and UndoIt()), we obtain additional information concerning typical reuse of the Command class through the corresponding association rules. (See Table 2, part (a).)

In particular, application classes that inherit from Command often override DoIt(), UndoIt(), TrackMouse(), and TrackFeedBack(). (See rules 1–4.) Yet, surprisingly, only 56% of application classes actually override DoIt() and UndoIt(). It turns out that other application classes reuse Command for its mouse tracking functionality by overriding TrackMouse() and TrackFeedBack().

However, of those application classes that do override DoIt(), 100% also override UndoIt() and vice versa. (See rules 5 and 6.) Moreover, observe that while 100% of application classes that override RedoIt() also override DoIt(), only 57% of application classes that override DoIt() also override RedoIt(). (See rules 7 and 8.)

Digressing for a moment, we consider the utility of knowing that certain member functions tend to be overridden when inheriting from a library class. After all, member functions like DoIt() and TrackMouse() are declared virtual in the header files so it is not suprising that they would be overridden in applications.

However, the whole point is not all virtual functions are overridden with equal frequency. If a member function is overridden most of the time — or under certain circumstances — then the developer should consider overriding the function in his own application under similar circumstances.

Moreover, our tool CodeWeb provides links into the corresponding application code, so a developer can also use the same applications used for data mining to also demonstrate how the library components are actually reused in practice. For example, it may not be clear what code to write to override the DoIt() and UndoIt() member functions until you see it done in several sample applications.

Now, returning to our example, let's examine the reuse patterns for the Application class. (See part (b), rules 1–8.) Specifically, we find that many classes that inherit from Application override DoMakeDocuments() and some also override About(), CanOpen(), DoMakeManager(), and ExtCommand(). Moreover, of those that override ExtCommand(), 100% also call ExtCommand() in the base class and vice versa. (See rules 7 and 8.) Examination of the source shows that any commands with which the application is not familiar are diverted by a call to the Application base class for further processing.

Finally, one can infer that developing 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. Again, the corresponding association rules also yield useful information. (See parts (c) and (d).)

For example, application components that inherit from Document often override DoMakeContent(), DoMakeMenuBar(), DoSetupMenu(), Control() and some also override CanLoad() and ReturnObjectToStore(). By examining the source code, one could infer, among other things, that DoMakeMenuBar() creates the applications menu bar, and that ReturnObjectToStore() returns that aspect of the model state that is to be stored on disk. Also, observe that 100% of application components that override CanLoad() also override ReturnObjectToStore() and vice versa. Again, one would see how to override these member functions by example in the sample applications.

While the reuse boundary intersection is a good place to start exploration, it is worthwhile to later examine the individual reuse boundaries for other useful library classes. For example, the library class Dialog is not present in the reuse boundary intersection but is reused by application Write and many other GUI-based applications. Since Dialog is typically reused many times in such applications, we find that the association rules inferred have higher support than the rules considered earlier in this section. (See part (e).)

Part	Association Rules	Conf.	Support
(a)	1. class_inherits:Command⇒member_function_overrides:DoIt()	56%	.59% (5)
	2. class_inherits:Command⇒member_function_overrides:UndoIt()	56%	.59% (5)
	3. class_inherits:Command⇒member_function_overrides:TrackMouse()	56%	.59% (5)
	4. class_inherits:Command⇒member_function_overrides:TrackFeedBack()	44%	.48% (4)
	5. member_function_overrides:DoIt()⇒member_function_overrides:UndoIt()	100%	.83% (7)
	6. member_function_overrides:UndoIt()⇒member_function_overrides:DoIt()	100%	.83% (7)
	7. member_function_overrides:RedoIt()⇒member_function_overrides:DoIt()	100%	.48% (4)
	8. member_function_overrides:DoIt()⇒member_function_overrides:RedoIt()	57%	.48% (4)
(b)	1. class_inherits:Application⇒member_function_overrides:DoMakeDocuments()	75%	1.1% (9)
	2. class_inherits:Application⇒member_function_overrides:About()	42%	.59% (5)
	3. class_inherits:Application⇒member_function_overrides:CanOpen()	33%	.48% (4)
	4. class_inherits:Application⇒member_function_overrides:DoMakeManager()	25%	.36% (3)
	5. class_inherits:Application⇒member_function_calls:ExtCommand()	25%	.36% (3)
	6. class_inherits:Application⇒member_function_overrides:ExtCommand()	25%	.36% (3)
	7. member_function_calls:ExtCommand()⇒member_function_overrides:ExtCommand()	100%	.71% (6)
	8. member_function_overrides:ExtCommand()⇒member_function_calls:ExtCommand()	100%	.71% (6)
(c)	1. class_inherits:Document⇒member_function_overrides:DoMakeContent()	100%	1.1% (9)
	2. class_inherits:Document⇒member_function_overrides:DoMakeMenubar()	89%	.95% (8)
	3. class_inherits:Document⇒member_function_overrides:DoSetupMenu()	67%	.71% (6)
	4. class_inherits:Document⇒member_function_overrides:Control()	56%	.59% (5)
	5. class_inherits:Document⇒member_function_overrides:CanLoad()	44%	.48% (4)
	6. class_inherits:Document⇒member_function_overrides:ReturnObjectToStore()	44%	.48% (4)
	7. member_function_overrides:CanLoad()⇒member_function_overrides:ReturnObjectToStore()	100%	.48% (4)
	8. member_function_overrides:ReturnObjectToStore()⇒member_function_overrides:CanLoad()	100%	.48% (4)
(d)	1. class_inherits:View⇒member_function_overrides:Draw()	86%	.71% (6)
	2. class_inherits:View⇒member_function_calls:GrPaintRect()	86%	.71% (6)
	3. class_inherits:View⇒class_instantiates:Point	71%	.59% (5)
	4. class_inherits:View⇒class_instantiates:Rectangle	71%	.59% (5)
	5. class_inherits:View⇒member_function_calls:ForceRedraw()	57%	.48% (4)
	6. class_inherits:View⇒member_function_calls:InvalidateRect()	57%	.48% (4)
	7. member_function_overrides:Draw()⇒member_function_calls:GrPaintRect()	75%	2.1% (18)
	8. member_function_calls:GrPaintRect()⇒member_function_overrides:Draw()	90%	2.1% (18)
(e)	1. class_inherits:Dialog⇒member_function_overrides:Control()	82%	1.7% (14)
	2. class_inherits:Dialog⇒member_function_calls:Control()	82%	1.7% (14)
	3. class_inherits:Dialog⇒member_function_instantiates:VObject	82%	1.7% (14)
	4. class_inherits:Dialog⇒member_function_overrides:DoMakeContent()	76%	1.5% (13)
	5. class_inherits:Dialog⇒member_function_overrides:DoSetup()	65%	1.3% (11)
	6. class_inherits:Dialog⇒member_function_calls:EnableItem()	65%	1.3% (11)
	7. class_inherits:Dialog⇒member_function_calls:ShowOnWindow()	65%	1.3% (11)
	8. class_inherits:Dialog⇒member_function_instantiates:Rectangle	53%	1.1% (9)

Table 2. Association rules for general reuse patterns. (Not all rules shown.)

5.2 Specialized Reuse Patterns

Generally speaking, looking at the reuse boundaries of a certain kind of applications can help identify library reuse patterns that are useful for other applications of this type. (Of course, we would also find patterns of more general utility also.)

For example, by inspecting the reuse boundaries for development tools DebuggerFW and ProgEnv, we can find reuse patterns particularly relevant to building other software development tools.

In particular, the reuse boundary intersection for DebuggerFW and ProgEnv contains classes CodeTextView, RegularExp, and PrettyPrinter. Moreover, ProgEnv additionally reuses classes CodeAnalyzer, AccessMembers, and AccessObjPtrs. Searching for association rules with these classes in the antecedent is helpful in identifying their characteristic reuse patterns.

For example, application classes that inherit from CodeTextView tend to override MakePrettyPrinter() (with 75% confidence and 0.36% support). Some examination of the source code would show that MakePrettyPrinter() determines which pretty printer to use (which is a subclass of the PrettyPrinter class mentioned above).

Finally, we should note that rules for specialized patterns tend to have lower support since, by definition, specialized library components are reused in fewer applications. Consequently, one must be more careful about the utility of such rules.

6 Related Work

Various tools have been described in the literature that help developers learn to use a software library. Excluding our previous work on reuse boundaries [17], we do not know of any other approaches that allow the user to select sample applications to demonstrate characteristic reuse of library components.

Data mining association rules in reuse boundaries improves upon our previous work by automatically identifying library components that are often reused in combination in applications. Inferring this information in a manual way from many large reuse boundaries is impractical — particularly if one is interested in the confidence and support of the association rules.

Other researchers have used data mining techniques in the software engineering domain but for different purposes. For example, data mining has been used to decompose a software system into data cohesive subsystems to assist developers with reengineering and maintenance tasks [8, 9]. As another example, a related technology, *machine learning*, has been used in reengineering class hierarchies [22] and generating test cases [5].

Perhaps closest to our approach is a technique for extracting specifications from software using machine learning [7]. In this work, instrumented code is run on a number of representative test cases, generating examples of its behavior. Inductive learning techniques are then used to generalize these examples, forming a general description of some aspect of the system's behavior. However, this research differs from our own in that the sample inputs are used to extract specifications while we use sample applications to discover characteristic reuse patterns.

In what follows, we shall describe some tools that are specifically designed to help developers learn to use a library and others that are more general in nature but are useful none-the-less useful for this purpose. As discussed elsewhere [12], such tools can be categorized as bottom-up or top-down. We follow this distinction in the following presentation of related work.

6.1 Bottom-up Approaches

Bottom-up approaches require that the user select and assemble architecturally compatible sets of components from a software library. For example, component retrieval tools are concerned with finding components that fit a particular need. Such work includes tools that use free-text indexing [11], facets [21], and specification matching [18]. Although these tools may help a developer find individual components of interest, they do not show how these components can be used in combination.

There are also tools that help a developer examine a software library in terms of architecture, style, etc. [6, 14, 26]. This may be helpful in understanding the design of the library which would facilitate reuse — particularly with frameworks. However, such tools require a bottom-up approach to understanding the essential components, their relationships, and collaborations.

These bottom-up approaches to learning to use a software library are akin to solving a Jigsaw puzzle [12]. It is not clear which components fit together and how.

6.2 Top-down Approaches

Top-down approaches start out with a subset of the software library that is particularly fundamental and typically reused in most applications. One can learn from this subset and adapt or extend it to build a new application.

For example, active cookbooks [20] and Microsoft Wizards guide the developer through common tasks (such as subclassing key classes) where the developer is asked to answer some questions, and skeleton code is generated automatically that reuses key library classes and provides a starting point for development.

However, such methods are only applicable for the tasks for which they were designed and may not help the developer's fundamental understanding of the library. Also, there is no guarantee that the developers of the library would foresee all typical applications of that library. Finally, these methods simply generate skeleton code; they do not present the user with examples on how to fill in and extend this skeleton code.

While there has been other research on the use of examples to illustrate software reuse [10, 19], the work most closely related with our own, as far as we know, is that on "exemplars" [12]. Specifically, 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. By browsing these classes as well as their static relationships and dynamic interactions, one can get a general understanding of how the framework works.

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.

While exemplars may be helpful, they are pre-selected and not representative of any particular class of applications. Moreover, they place an extra burden on the developers of the software library. In contrast, our approach allows the user to examine a particular aspect of the library that is of interest. Moreover, the tool is fully automated and works on any existing code.

Finally, we note that our approach is also top-down. One starts out with some fundamental library components (as inferred by inspecting the reuse boundaries) and then queries/browses association rules for such components.

Returning to the Jigsaw puzzle analogy, observe that puzzle pieces are designed to fit an outline. Consequently, it would be easier to start out with the outline and place pieces accordingly. Top-down approaches identify those essential components in the library that one can extend and modify in an application — in essence, the outline to the Jigsaw puzzle.

7 Conclusions and Future Work

In this paper, we have shown how data mining can be used to discover library reuse patterns in user-selected applications. This can be helpful in building and debugging applications that use a particular library by observing how other developers have used that library in their applications. As an example, we have shown how a developer might use our tool, CodeWeb, to identify reuse patterns in the ET++ application framework.

In our approach, we mine *association rules* that identify library components that are often reused in combination by application components. By querying and/or browsing such association rules, a developer can discover patterns for reusing library components. Moreover, the association rules can be used to automatically warn developers when their application reuses library components in a way that differs from characteristic usage in applications written by others.

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 software library. Also, it is important to note that our method is not intended to replace current techniques for learning to use libraries but, rather, to complement them.

While we have used the ET++ framework as a running example throughout this paper, it is actually not the best candidate to demonstrate data mining. While ET++ is elegant and well-known in the research literature (which is partly why we used it here), few applications have been written using it.

Our data mining approach works better with more applications. We do not see this as a major shortcoming since most developers are interested in learning those libraries that are widely used in practice. Currently, we are carrying out a case study using the popular KDE libraries and over one hundred real-life applications based on them.[1]

In this paper, we mined association rules at a reasonably high level of abstraction by considering classes, functions, and various reuse relationships between. For some relationships, such as method calls, we have completely abstracted away the order in which they occur. In future work, it would be interesting to take into account ordering when generating association rules. For example, with an I/O library, one might find that files are typically processed by calling functions open(), read()/write(), and close() in that order.

Currently, we extract all information statically from the source code. However, it may be interesting to incorporate dynamic information from running applications that use a particular library. For example, when determining method call relationships, we may include only those that occur with a high frequency. In this way, it may be possible to abstract away non-essential calls thus focusing the user's attention to particularly fundamental association rules.

Finally, one can view the approach described in this paper as learning from *positive experience*. That is, library reuse that has worked in practice. (Presumably, one would select "stable" applications to demonstrate reuse patterns in

[1]The KDE project aims to build a graphical desktop environment for Unix workstations that rivals Microsoft Windows. See http://www.kde.org for details. We have mined KDE reuse patterns by analyzing over 100 applications. The resulting association rules are available at http://www.cs.washington.edu/research/projects/se/www/kde/reuse_patterns.

a library.) However, one can also mine *negative experience*. That is, misunderstandings and problems that came up when reusing components from a software library. For future work, it would be interesting to determine if one can mine negative experience in an automated way — perhaps by analyzing application CVS logs for reuse patterns that were problematic and later corrected.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, pages 487–499, 1994.

[3] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD '98*, pages 85–93, 1998.

[4] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *15th International Conference on Data Engineering*, 1999.

[5] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Transactions on Software Engineering and Methodology*, 5(2):119–145, 1996.

[6] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[7] W. W. Cohen. Inductive specification recovery: Understanding software by learning from example behaviors. *Automated Software Engineering*, 2(2):107–129, 1995.

[8] C. Montes de Oca and D. L. Carver. Identification of data cohesive subsystems using data mining techniques. In *Proceedings of the International Conference on Software Maintenance*, pages 16–23, 1998.

[9] C. Montes de Oca and D. L. Carver. A visual representation model for software system decomposition. In *Proceedings of the 5th International Working Conference on Reverse Engineering*, pages 231–240, 1998.

[10] G. Fischer, S. Henninger, and D. Redmiles. Intertwining query construction and relevance evaluation. In *CHI '91*, pages 55–62, 1991.

[11] 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.

[12] D. Gangopadhyay and S. Mitra. Design by framework completion. *Automated Software Engineering*, 3:219–237, 1996.

[13] J. Gessaroli. Data mining: A powerful technology for database marketing. *Telemarketing*, 13(11):64–68, 1995.

[14] R. Kazman and S. J. Carriere. View extraction and view fusion in architectural understanding. In *5th International Conference on Software Reuse*. IEEE, 1998.

[15] C. D. Krivda. Booming business intelligence. *Midrange Systems*, 8(12):32–34, 1995.

[16] C. D. Krivda. Unearthing underground data. *LAN*, May 1996.

[17] 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*, pages 200–203, 1998.

[18] A. Moormann-Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.

[19] L. R. Neal. A system for example-based programming. In *CHI '89*, pages 63–68, 1989.

[20] W. Pree, G. Pomberger, A. Schappert, and P. Sommerlad. Active Guidance of Framework Development. *Software-Concepts and Tools*, 16(3):136–45, 1995.

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

[22] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *6th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 99–110, 1998.

[23] S. Sparks, K. Benner, and C. Faris. Managing object-oriented framework reuse. *Computer*, 29(9):52–61, 1996.

[24] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st VLDB Conference*, 1995.

[25] A. Weinand, E. Gamma, and R. Marty. ET++—an object oriented application framework in C++. In *OOPSLA*, pages 46–57, 1988.

[26] 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.