Data Mining Library Reuse Patterns using Generalized Association Rules

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 existing applications. Specifically, we consider the problem of discovering library classes and member functions that are typically reused in combination by application classes. This paper improves upon our earlier research using "association rules" [8] by taking into account the inheritance hierarchy using "generalized association rules". This turns out to be a non-trivial but worthwhile endeavor.

By browsing generalized association rules, a developer can discover patterns in library usage in a way that takes into account inheritance relationships. For example, such a rule might tell us that application classes that inherit from a particular library class often instantiate another class *or one of its descendents*. We illustrate the approach using our tool, CodeWeb, by demonstrating characteristic ways in which applications reuse classes in the KDE application framework.

Keywords
Software libraries, reuse patterns, data mining.

1 INTRODUCTION

Using a software library is not easy. Selecting the right components for reuse is only part of the problem. There is still the issue of reusing these components in the right way. This is particularly important with application frameworks where multiple components are reused in combination. A classic example is Smalltalk's Model/View/Controller.

Indeed, Will Tracz writes:

> If you have components to reuse, then you need to glue them together... After you glue pieces together long enough, you start seeing a pattern, then you can reuse the glue too [12, p. 41].

ICSE 2000 Limerick Ireland

In this paper, we shall discover patterns in the ways library classes are reused in practice, which we call *reuse patterns*. This is done by data mining existing applications that use the library. For example, we may find that most application classes that inherit from a library class Widget tend to override its member function paint(). As another example, we may notice that most application classes that instantiate a library class Painter and that call its member function begin() also call its member function end().

Traditionally, such knowledge is presented by example in library tutorials and/or toy programs. However, not all libraries come with such representative examples of reuse. This is particularly true for libraries developed by a company for internal use only and libraries developed by the open source community. Writing quality tutorials and toy programs is a time consuming and difficult process. Moreover, software developers would rather write code than documentation.

Our reuse pattern approach has the following benefits: (1) by leveraging existing applications and using data mining technology, we do not need expert analysis to identify characteristic usage of the library; (2) by using many real-life applications instead of a few toy programs, we can demonstrate reuse of many library classes in numerous and deeper ways; and (3) by using automated techniques, we can keep reuse patterns up to date with respect to the most recent version of the library and applications.

This paper improves upon our earlier research on reuse patterns using "association rules" [8] by taking into account the inheritance hierarchy using "generalized association rules". By browsing generalized association rules, a developer can discover patterns in library usage in a way that takes into account inheritance relationships. For example, such a rule might tell us that application classes that inherit from a particular library class often instantiate another class *or one of its descendents*.

The paper is organized as follows. Section 2 introduces the field of data mining and describes a well-known method for addressing the "shopping basket analysis" problem that takes into account taxonomies. Section 3 demonstrates how a similar data mining technique can be used to discover software

library reuse patterns in existing applications in a way that takes into account inheritance relationships. Section 4 shows how browsing such reuse patterns for the KDE application framework can illustrate characteristic reuse in existing applications. Section 5 discusses related work. Section 6 summarizes the paper, concluding with future work.

2 DATA MINING

Data mining may be defined as follows:

> The process of nontrivial extraction of implicit, previously unknown and potentially useful information (such as knowledge rules, constraints, regularities) from data [9].

Data mining is widely used in business to gain a competitive edge. An effective data mining application in the retail environment is *shopping basket analysis*. Progress in barcode technology has made it possible to store *basket data* that contains items purchased on a per-transaction basis. By using data mining technology, one can find patterns in items that are bought in combination.

Association Rules

Shopping basket analysis can be done by mining *association rules* [1]. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. 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 $X \cap Y = \emptyset$. (Throughout this paper, we shall distinguish sets of items from individual items by using upper and lower case letters, respectively.)

For example, suppose that people who purchase bread and butter also tend to 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.

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.

Confidence and Support

Let D be a set of transactions, where each transaction T is a set of items (*itemset*) such that $T \subseteq I$. We say that a rule $X \Rightarrow Y$ holds in transaction set D with *confidence* $c\%$ if $c\%$ of transactions in D that contain X also contain Y. We say that itemset Z has *support* $s\%$ in transaction set D if $s\%$ of transactions in D contain Z. Support is also defined for rules: $X \Rightarrow Y$ has *support* $s\%$ if itemset $X \cup Y$ has support $s\%$. For brevity, we may at times write "supp(Q)" to indicate the support of an itemset or rule Q in D.

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.

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 transaction, 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, support is a measure of statistical significance in the following sense: if $\text{supp}(X \Rightarrow Y) \approx \text{supp}(X) \times \text{supp}(Y)$, then it is likely that X and Y are independent and co-occur in transactions by chance; however, if $\text{supp}(X \Rightarrow Y) \gg \text{supp}(X) \times \text{supp}(Y)$, then this is not likely to be the case [7]. (We make this argument more precise in Section 3.) Returning to our example, it may be that $a \Rightarrow c$ is more statistically significant although $a \wedge b \Rightarrow c$ has higher confidence.

Association Rule Mining Problem

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

> Generate all association rules that have support $s\%$ at least as great as some user-specified minimum support $s_{\min}\%$ and confidence $c\%$ at least as great as some user-specified minimum confidence $c_{\min}\%$.

Several algorithms have been presented in the literature for finding all such association rules. Many of them are variations on the *Apriori* algorithm [1], which works in two phases: (1) it finds all itemsets that have support above the minimum support; and (2) it uses these itemsets to generate all rules whose confidence is above the minimum confidence. In such an algorithm, larger values of $s_{\min}\%$ can reduce the running time significantly but larger values of $c_{\min}\%$ have little effect on the running time (although they do yield fewer rules of course).

Taxonomies

In this paper, we shall be concerned with *generalized association rules* which take into account the presence of taxonomies [11]. By taxonomies, we mean "is-a hierarchies" where a node's descendents represent specializations of that

node.

For example, a taxonomy may indicate that white bread is a kind of bread and that skim milk is a kind of milk. In that case, the taxonomy would have white bread as a descendent of bread and skim milk as a descendent of milk. As we shall see in Section 3, such taxonomies allow us to take into account the class inheritance hierarchy when mining for library reuse patterns.

Taxonomies allow us to mine for rules at different levels of abstraction. This is important since interesting associations among data items often occur with more abstract concepts. In particular, purchase patterns may not show any substantial regularities at the primitive data level, but may show some interesting regularities at higher levels of abstraction.

For example, the rule "white bread∧butter⇒skim milk" may have insufficient support using standard association rule mining, but "bread∧butter⇒milk" may pass the support requirement using generalized association rules. That is because additional transactions may support other kinds of bread and milk.

Formally speaking, we model one or more taxonomies as a directed acyclic graph $\mathcal{T}$ on the items $I = \{i_1, i_2, \ldots, i_m\}$. Edges in the $\mathcal{T}$ denote is-a relationships among items. Specifically, an edge from c to p in $\mathcal{T}$ indicates that p is the parent of c and means that c is a particular kind of p (or in other words, that p is a generalization of c.)

Generalized Association Rules

Generalized association rules improve upon standard association rules by incorporating a taxonomy $\mathcal{T}$. In particular, a generalized 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 as $X \Rightarrow Y$, where $X \subset I, Y \subset I, X \cap Y = \emptyset$, and no item in Y is an ancestor of any item in X. The reason for this latter requirement is that any rule of the form "$x \Rightarrow$ ancestor(x)" is true with 100% confidence and consequently redundant.

Before proceeding further, we first define a partial order on itemsets using the taxonomy $\mathcal{T}$. In particular, $X < Y$ if and only if we can get from itemset Y to itemset X by replacing one or more items in Y with some ancestor(s) in $\mathcal{T}$. Observe that X may have fewer or greater items than Y; for example {food} < {bread, butter} and {dairy product, liquid} < {milk}. If $X < Y$, then itemset X is an *ancestor* of Y or, equivalently, Y is a *descendent* of X.

Now, we say that a generalized association rule $X \Rightarrow Y$ holds in transaction set D with *confidence* c% if c% of transactions in D that contain X, or a descendent of X, also contain Y, or a descendent of Y. Moreover, itemset Z has *support* s% in transaction set D if s% of transactions in D contain Z or a descendent of Z. Support for rules is defined as follows: $X \Rightarrow Y$ has *support* s% if the itemset $X \cup Y$ has support s%.

Our earlier observations with respect to confidence and support still apply in this more general context. Moreover, if a rule $X \Rightarrow Y$ has minimum support and confidence, then the rule $X \Rightarrow$ ancestor(Y) is guaranteed to have both minimum support and confidence also. However, the rules ancestor(X) $\Rightarrow Y$ and ancestor(X) $\Rightarrow$ ancestor(Y) have minimum support but not necessarily minimum confidence.

To support taxonomies, one can use algorithms for mining standard association rules by considering "extended transactions" that contain not only the items in transactions but also their ancestors. To make this process efficient, certain optimizations are done to restrict the number of itemsets that need to be counted at various stages in the algorithm and the number of ancestors added to form extended transactions [11].

Finally, there is also the problem that taxonomies tend to yield more rules, many of which are redundant. Consequently, pruning the output to show only "interesting rules" is essential. We shall describe our pruning technique in Section 3.

3 MINING REUSE PATTERNS

Now that we have introduced generalized association rule mining in Section 2, we demonstrate how to apply this technology to discover library reuse patterns in existing applications. We do this in a way analogous to that for discovering items that are typically purchased together in basket data. Specifically, we identify library classes and member functions that are often reused in combination by application classes.

For example, we may find that application classes that inherit from the library class Widget usually override its member function Widget::paint(). In that case, we would generate the generalized association rule:

class_inherits:Widget ⇒
class_overrides:Widget::paint().

Of course, one might question the utility of such a rule. It may be obvious from the library source that paint() could be overridden in application classes (perhaps because it is declared virtual in a language like C++). However, 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.

Applying Generalized Association Rule Mining

Conceptually, our application of generalized association rule mining in the manner described above is simple. However, in practice, there are many issues to consider. For example, what kinds of reuse relationships do we want to include? How do we take advantage of the inheritance hierarchy? How do we prune the resulting rules?

In what follows, we present a fairly detailed account of applying our approach to C++ software libraries and applications. In particular, we define our notion of items $I = \{i_1, i_2, \ldots, i_m\}$, explain how to construct the set of transactions D, and present a taxonomy $\mathcal{T}$ over items I based on the class inheritance hierarchy.

While existing data mining algorithms can be used [11], there are issues to consider while pruning the resulting generalized association rules. Thus, we conclude our discussion by describing the pruning process.

Items

In our use of generalized association rules, items indicate reuse relationships involving classes or member functions. For example, a typical item might be class_inherits:Widget where the reuse relationship is inheritance and the class reused is Widget. As another example, the item class_overrides:Widget::paint() indicates an overriding relationship with Widget's member function paint().

We associate with every application class A the set of all items I(A) that are involved in a reuse relationship with class A. Returning to our example, if a class myWidget (only) inherits from Widget and overrides paint(), then

I(myWidget)={class_inherits:Widget, class_overrides:Widget::paint()}.

Also, the reuse relationship need not involve a library class. For example, if myDialog instantiates myButton, then class_instantiates:myButton$\in I$(myDialog).

The complete set of items I is defined as $I = \bigcup_A I(A)$ where the union is over all application classes A. The set I contains items with reuse relationships that involve both application and library classes.

In what follows, we shall expand on the various reuse relationships. Before doing so, we extend our notation to include library and application names as prefixes of classes and their member functions. For example, we shall write app'myDialog to indicate that myDialog is defined in application app. Similarly, we shall write lib'Widget::paint() to indicate that member function Widget::paint() is defined, or at least declared as an abstract member, in library lib. Global functions, such as main(), are represented by omitting the class name, as with app'main().

There are five reuse relationships that we have considered in our research: class inheritance, class instantiation, function invocation, function overriding, and implicit invocation. Although our approach can be applied to any object-oriented programming language, we shall, for concreteness, base our presentation on C++. For the reuse relationships mentioned, we present the corresponding item types below:

class_inherits:p'class This item expresses a standard inheritance relationship. Both single and multiple inheritance are allowed.

class_instantiates:p'class We say that class or member function A *instantiates* class B if and only if (1) A allocates an instance of B on the stack by way of a non-pointer variable of type B or (2) A allocates an instance of B on the heap using new or malloc(). Either way, A instantiates a new instance of B. We only generate instantiation items for classes. So, if A is actually a member function of some class C, then we say that C instantiates B and write this as the item class_instantiates:p'B.

class_calls:p'class::func() If a function p'A::f() calls a function q'B::g(), then we construct an item indicating a call made by class A: class_calls:q'B::g()$\in I$(p'A).

class_overrides:p'class::func() If a class p'A inherits from q'B and overrides its member function q'B::f(), then we construct the item class_overrides:q'B::f().

class_receives_signal:p'class::signal() If implicit invocation is used in the libraries and applications at hand, we assume there is some way to easily identify how this is done through simple lexical analysis of the source. If an object of type A broadcasts "signal()" that is received by an object of type B, then we construct an item class_receives_signal:p'A::signal()$\in I$(q'B). Moreover, we also include a class_calls item for the emission by A.

For each of the items defined above, there is also a corresponding version with a '^' symbol appended to the item class. Such items are used whenever the corresponding application class may reuse a descendent of the class in the item. For example, if an application class is associated with class_instantiates:lib'PushButton, then we also associate the item class_instantiates:lib'Button^ since PushButton inherits from Button.

Transactions

We could simply define each transaction T(A) as equal to I(A) for each A. That is, simply include all items associated with each application class. We could then prune uninteresting rules after the data mining is complete. However, this would be very inefficient. It pays to reduce the number of items in transactions as early as possible to reduce computational blowup in the mining process.

Recall that our goal is to identify how library classes are typically reused in existing applications. Moreover, we do not care so much about how library classes reuse other library classes nor how application classes reuse other application classes. Rather, we are interested in how application classes reuse library classes.

One might try to construct transactions consisting of those items associated with application classes that refer to library classes and functions. For example, we could include an item like class_inherits:lib'Widget in a transaction for application class app'myWidget but exclude any items referring to application classes or functions.

However, there is a problem in doing this. Consider a class app'myDialog that inherits from lib'Dialog and instantiates app'myPushButton that in turn inherits from lib'PushButton. In such a case, we would like to include class_instantiates:app'myPushButton in app'myDialog's transaction so that we may (indirectly) support the following generalized association rule:

class_inherits:lib'Dialog ⇒
class_instantiates:lib'PushButton^.

Also, suppose app'myDialog makes a call to app'myPushButton::show(), where show() is defined in PushButton's ancestor Widget. In that case, we would like to include class_calls:app'myPushButton::show() in app'myDialog's transaction so that we may (indirectly) support the following generalized association rule:

class_inherits:lib'Dialog ⇒
class_calls:lib'PushButton^::show().

Consequently, we include additional items in transactions to make this possible. In particular, given the items $I(\mathsf{A})$ for an application class A, we include in $T(\mathsf{A})$ only the following items in $I(\mathsf{A})$: (1) all items involving a library class lib'A or member function lib'A::f(); (2) those items involving application classes app'A that inherit directly or indirectly from a library class lib'B; and (3) those items involving application member functions app'A::f() where class app'A inherits directly or indirectly from a library class lib'B that defines or overrides member function f.

Taxonomy

We define a taxonomy $\mathcal{T}$ in a way that mirrors the inheritance hierarchy. Consider the set of all transaction items I_{trans} defined as $I_{\text{trans}} = \bigcup_A T(A)$ where the union is over all application classes A and $T(A)$ denotes the items in the transaction for A. For efficiency, we construct a taxonomy that is induced by only those items in I_{trans}.

First, we consider items in I_{trans} that do not involve member functions. These denote inheritance or instantiation. Without loss of generality, we consider instantiation; inheritance is handled similarly. For an item class_instantiates:p'C$\in I_{\text{trans}}$, we consider all ancestors $\mathsf{A}_1, \ldots, \mathsf{A}_k$ of C in the inheritance hierarchy.

The item class_instantiates:p'C contributes the following to the taxonomy $\mathcal{T}$: (1) nodes class_instantiates:p'C, class_instantiates:p'C^, class_instantiates:p_1'A_1^, ..., class_instantiates:p_k'A_k^; (2) an edge from class_instantiates:p'C to class_instantiates:p'C^; (3) edges from class_instantiates:p'C^ to class_instantiates:p_i'A_i^ where C inherits from A_i; and (4) edges from class_instantiates:p_i'A_i^ to class_instantiates:p_j'A_j^ where A_i inherits from A_j.

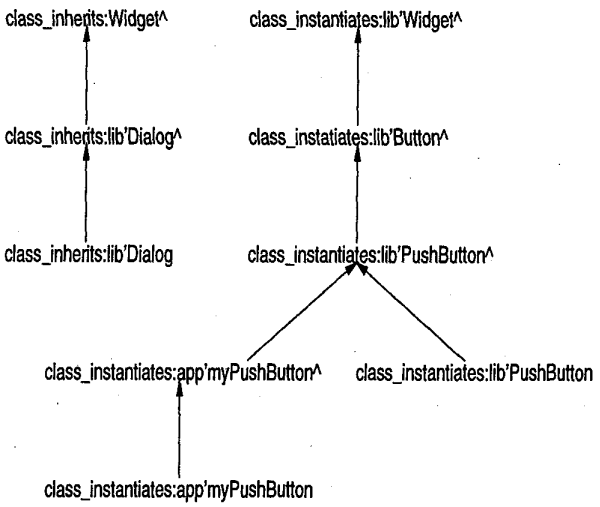


Figure 1: Items not involving member functions in taxonomy.

By distinguishing items using '^', a user can tell whether a rule involves only the items mentioned or possibly their descendents. For example, a rule involving item class_instantiates:p_i'A_i^ makes it very clear that some application classes that support the rule may instantiate a descendent of A_i. However, if the rule had class_instantiates:p_i'A_i, then all application classes that support this rule instantiate A_i. Finally, observe that the mining algorithm may generate both kinds of rules with different confidence and support. Browsing both can be helpful in learning to use the library.

Let us again consider a class app'myDialog that inherits from lib'Dialog and instantiates app'myPushButton that in turn inherits from lib'PushButton. The transaction for app'myDialog contains items class_inherits:lib'Dialog and class_instantiates:app'myPushButton. Figure 1 shows the contributions made by these two items to the taxonomy $\mathcal{T}$. Observe that another transaction has contributed the node class_instantiates:lib'PushButton and the edge from class_instantiates:lib'PushButton to class_instantiates:lib'PushButton^.

Next, we consider items in I_{trans} that *do* involve member functions. These denote function invocation, function overriding, or implicit invocation. We handle such items in much the same way as those not involving member functions except that we only consider ancestor classes that inherit or define the member function in question.

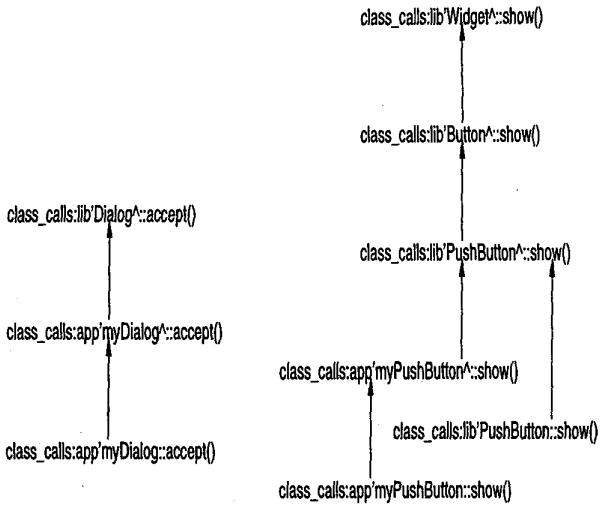


Figure 2: Items involving member functions in taxonomy.

For example, suppose a class app'myDialog calls app'myDialog::accept() and app'myPushButton::show(). The transaction for app'myDialog contains items class_calls:app'myDialog::accept() and class_calls:app'myPushButton::show(). Figure 2 shows the contributions made by these two items to the taxonomy $\mathcal{T}$. Observe that another transaction has contributed the node class_calls:lib'PushButton::show() and the edge from class_calls:lib'PushButton::show() to class_calls:lib'PushButton^::show().

Pruning

We do not present all generalized association rules to the user but only those that we deem "interesting". This is particularly important since, in practice, we find that there are thousands of similar and/or redundant rules that would overwhelm the user otherwise.

We shall assume that there is only one item in the consequent of each generalized association rule. This is not a serious limitation since $X \Rightarrow y_1 \wedge y_2 \wedge \cdots \wedge y_n$ is logically equivalent to $(X \Rightarrow y_1) \wedge (X \Rightarrow y_2) \wedge \cdots \wedge (X \Rightarrow y_n)$; so, instead of showing a rule with n items in the consequent to the user, we instead show n rules each having a consequent of one item.

Global Pruning

Initially, we perform a global pruning that is applied to all rules discovered by data mining. Afterwards, we shall perform local pruning on various subsets of rules that survive global pruning.

Uninteresting Rules We prune rules such as the following:

class_calls:lib'A::f() ⇒ class_instantiates:lib'A

Such a rule is hardly surprising. It is quite likely that any reuse of a class member function implies that the class is inherited from or instantiated at some point. Consequently, we prune rules where an item in the antecedent involves a member function lib'A::f() and an item in the consequent instantiates or inherits a class B — where B may be A, an ancestor of A, or a descendent of A — that defines or inherits member function f().

In addition, we prune rules with a class in the antecedent that is a (strict) descendent or ancestor of the class in the consequent. For example, a rule such as class_instantiates:lib'A^ ⇒ class_calls:lib'B::f() is pruned if and only if A is a descendent of B or B is a descendent of A. The rationale behind this heuristic is to avoid the numerous rules that would result in a deep inheritance hierarchy (e.g., if say A has many descendents B).

Misleading Rules Some generalized association rules are misleading [2]. For example, suppose the rule $x \wedge y \Rightarrow z$ has confidence 60% while the rule $y \Rightarrow z$ has confidence 80%. In that case, the first rule is misleading since the presence of x actually *decreases* the likelihood of finding the item z. More generally, given a rule $X \Rightarrow y$ with confidence $c\%$, we say it is *misleading* if and only if there is another rule $X' \Rightarrow y$ with confidence $c'\%$ where $\emptyset \subseteq X' \subset X$, $c\%/c'\% < \gamma$, and where γ is a user-defined threshold typically set at one or over. (In this case, we allow $X' = \emptyset$ in this other "rule" in which case the confidence is simply the support of item y.)

Statistically Insignificant Rules We can view the support of an itemset X as the probability of finding X in a transaction. A rule $X \Rightarrow Y$ is not very interesting to us if X and Y just happen to co-occur in transactions by chance. Specifically, if $\text{supp}(X \Rightarrow Y) \approx \text{supp}(X) \times \text{supp}(Y)$, then it is likely that X and Y are independent and $X \Rightarrow Y$ would not be statistically significant. Yet, if $\text{supp}(X \Rightarrow Y) \gg \text{supp}(X) \times \text{supp}(Y)$, then this is not likely to be the case.

More specifically, suppose we find that X and Y have supports p_x and p_y which we shall use as estimates for the true probabilities of finding the respective itemsets in any given transaction. If we assume that X and Y are independent, then the probability of finding both in a particular transaction is $p_x p_y$. Now, suppose we have n transactions and the rule $X \Rightarrow Y$ occurs in k of them (with support k/n). Then the probability that $X \Rightarrow Y$ occurs in k or more transactions assuming that X and Y are independent is [7]:

$$\sum_{i=k}^{n} \binom{n}{i} (p_x p_y)^i (1 - p_x p_y)^{n-i}.$$

Clearly, the lower the above probability — the so-called *p-value* — the more statistically significant the rule $X \Rightarrow Y$ since it is less likely that X and Y are independent. In our experiments, we ensure that all rules have p-values of λ or less, where λ is a user-specified threshold.

Local Pruning

In addition to the global pruning described above, we also prune locally on various subsets of the rules discovered. Specifically, our tool allows the user to browse: (1) rules that demonstrate reuse of a particular library class; and (2) rules that are violated in a particular application (where the tool acts like a "reuse lint").

In the former case, we consider only the set of rules with that library class in the antecedent or consequent. In the latter case, we consider only those rules that are violated by at least one class in the application of interest. Either way, once we have extracted the subset of rules, we follow the same local pruning procedure when presenting the results to the user.

The motivation for local pruning is the following: given the presence of a particular rule, another rule may not be surprising to us. In that case, it is desirable to additionally prune the latter rule to focus the user's attention on those rules that are interesting. The pruning process that follows builds upon several existing techniques [2, 11].

Consider rules $X \Rightarrow y$ and rule $X' \Rightarrow y$, where X' is a subset of X. If we know the confidence $c'\%$ for rule $X' \Rightarrow y$, then we expect the confidence for rule $X \Rightarrow y$ to also be $c'\%$ since there is no reason to believe — without prior knowledge of the library and/or applications — that the additional items $X - X'$ in the antecedent are likely to increase/decrease the occurrence of y. Thus, we shall consider pruning $X \Rightarrow y$ if its confidence $c\%$ is not much greater than $c'\%$. More specifically, we set an *interest threshold* δ and prune any such rule $X \Rightarrow y$ where $c\%/c'\% < \delta$. If it is not pruned, we say that $X \Rightarrow y$ is *δ-interesting* with respect to $X' \Rightarrow y$.

We also perform another form of pruning. Suppose we have two rules $X \Rightarrow y$ and $\hat{X} \Rightarrow \hat{y}$ where $\hat{X}$ is an ancestor of itemset X containing the same number of items (where "ancestor" means $\hat{X} < X$ as defined in Section 2) or $\hat{y}$ is an ancestor of y (or both). In such a case, we may keep the more specific rule $X \Rightarrow y$ and prune $\hat{X} \Rightarrow \hat{y}$.

Generally speaking, we would like to show the more specific rule which tends to be more informative. However, we may also wish to show the more general rule if its confidence is much greater than expected.

If $X \Rightarrow y$ has confidence $c\%$, then $\hat{X} \Rightarrow y$ has expected confidence $c\%$ since there is no reason to believe, without prior knowledge, that y is more/less likely to be in a transaction with $\hat{X}$ than one with X. However, as $\hat{y}$ is an ancestor of y, the rule $X \Rightarrow \hat{y}$ clearly has confidence $c\%$ and possibly much more. To get a reasonable estimate for expected support, we shall assume prior knowledge of the relative support of y and $\hat{y}$. With such knowledge, we would expect $X \Rightarrow \hat{y}$ to have confidence $(\text{supp}(\hat{y})/\text{supp}(y)) * c\%$ since of those transactions that support X, we would expect $\text{supp}(\hat{y})/\text{supp}(y)$ of them to support $\hat{y}$.

By the first observation, $\hat{X} \Rightarrow y$ has expected confidence $c\%$. Given the confidence $c\%$ for $\hat{X} \Rightarrow y$, we would expect $\hat{X} \Rightarrow \hat{y}$ to have confidence $(\text{supp}(\hat{y})/\text{supp}(y)) * c\%$ as explained above. Consequently, we prune the rule $\hat{X} \Rightarrow \hat{y}$ if and only if $\frac{\hat{c}\%}{(\text{supp}(\hat{y})/\text{supp}(y)) * c\%} < \delta$ for some interest threshold δ.

Finally, combining this analysis with our earlier results on smaller antecedents X', we shall prune a rule $\hat{X} \Rightarrow \hat{y}$ given confidence $c'\%$ for $X' \Rightarrow y$ if and only if $\frac{\hat{c}\%}{(\text{supp}(\hat{y})/\text{supp}(y)) * c'\%} < \delta$. This follows because the expected confidence for $X \Rightarrow y$ is the same as the confidence $c'\%$ for $X' \Rightarrow y$.

Now, we are ready to describe the complete pruning procedure. Given a set of rules $\{X_1 \Rightarrow y_1, \ldots, X_n \Rightarrow y_n\}$, we first construct a partial order with a node for each rule. The nodes in the partial order are ordered as follows: $X_i \Rightarrow y_i < X_j \Rightarrow y_j$ if and only if: (1) the rules are not identical; (2) $\emptyset \subseteq X_i \subseteq X_j$; and (3) X_i is more specific than or equal to X_j and y_i is a descendent of or equal to y_j.

Pruning proceeds by considering the nodes of the partial order in topological sort order. That is, an ancestor is always processed before its descendents. In this process, a rule $X \Rightarrow y$ is not pruned if and only if it is δ-interesting with respect to all ancestors in the partial order that have survived the pruning process to that point.

4 BROWSING REUSE PATTERNS

In this section, we shall demonstrate how one might browse and learn from generalized association rules by considering code written for the KDE desktop environment. The KDE libraries provide a C++ application framework for developing GUI applications. In our experiment, we have mined reuse patterns for the KDE 1.1.2 core libraries (which include the Qt toolkit) by analyzing 76 real-life applications.

Specifically, we have used our tool, CodeWeb, to mine for generalized association rules with confidence of at least 10% and support of at least 15 transactions. (There were 1365 transactions total so the support requirement as a percentage is about 1.1%.) The global pruning parameters were set at $\gamma = 1.25$ and $\lambda = 0.01$. The local pruning parameter δ was set to 1.25. Only rules with one item in the antecedent and one item in the consequent were considered.

To contrast generalized association rule mining with our earlier work on standard association rule mining, we also include the data mining results where inheritance was ignored all together. On a Sparc Ultra 1, mining generalized association rules took about 50 minutes while mining standard

association rules took 20 minutes. The rule statistics are as follows:

	Rules	Global Pruning			Rules
	Mined	Uninteresting	Misleading	Insignificant	Left
Generalized	51308	13299	1904	6681	34271
Standard	21594	996	320	978	19636

Observe that while the number of rules mined by generalized association rules is significantly more than that using standard association rules, a greater percentage of these rules is eliminated during global pruning. Of course, the number of rules pruned locally varied depending upon the local context and is not shown here.

Typically, a developer just starting out with a library would identify important library classes and browse their reuse patterns. By "important", we mean those library classes that are reused in many existing applications and are thus likely to be relevant in new applications also. For example, a developer using our tool would notice that the KDE classes KApplication and QObject are reused in 99% and 100%, respectively, of the 76 applications mined — and are thus essential in any KDE application. We consider the reuse patterns for these two classes in what follows.

KApplication Reuse Patterns

Figure 3 shows all reuse patterns predicated on the instantiation of KApplication in an application class. The *supporters* of a rule are those application classes for which all rule items apply. For example, an application class that supports reuse pattern #1 must instantiate KApplication and calls its member function exec(). We also show the *detractors* of a rule which are those application classes for which the antecedent items apply but where the consequent item does not hold. For example, an application class that detracts from reuse pattern #1 must instantiate KApplication and *not* call exec(). Our tool allows users to browse the source code for both supporters and detractors of reuse patterns; these application classes illustrate characteristic and uncharacteristic reuse, respectively.

In Figure 3, we find — among other things — that of those applications classes that instantiate the KApplication class, 72.3% call its member function exec(), 58.5% instantiate KTopLevelWidget^, 53.8% call the member function setMainWidget() of the class KApplication, and 46.2% call the show() member of the class KTopLevelWidget^. Recall that the symbol '^' indicates that some application class may reuse a strict descendent of KTopLevelWidget rather than the class itself.

By browsing reuse patterns in combination with library reference documentation (which is usually available) and application source code, a developer can learn to use a library by example in much the same way as studying manually constructed tutorials and/or toy programs (both of which may not be available).

For example, doing this for the rules in Figure 3 reveals that the class KApplication, instantiated in the main() function of most KDE applications, manages the application event queue. We also observe that applications inherit from KTopLevelWidget to define the main widget of the application (e.g., one that is not contained in any other); this widget is then instantiated and a call to setMainWidget() tells the library that whenever the user closes this widget that the application should terminate all together. Afterwards, the exec() member of KApplication is called to enter the main event loop.

Finally, it turns out that all applications that instantiate KTopLevelWidget^ always instantiate a descendent that they define. Without taking into account the inheritance hierarchy (e.g., using generalized association rules), we would miss reuse patterns involving this class all together.

QObject Reuse Patterns

The class QObject is an ancestor of almost all classes in the KDE libraries. According to the reference documentation, this class provides facilities for event handling and timing operations. Figure 4 shows some of the reuse patterns reported by our tool for QObject; of the 53 rules found for this class, 47 involve a '^' symbol in the antecedent and/or consequent. It turns out that application classes rarely reuse QObject directly; typically, they reuse it indirectly through one of its descendents. Although QObject is very fundamental to the KDE libraries, only six rules would have been found without taking into account the inheritance hierarchy.

5 RELATED WORK

In this paper, we have looked for patterns in the way library classes have been reused in practice by existing applications. In this section, we shall talk about several related techniques.

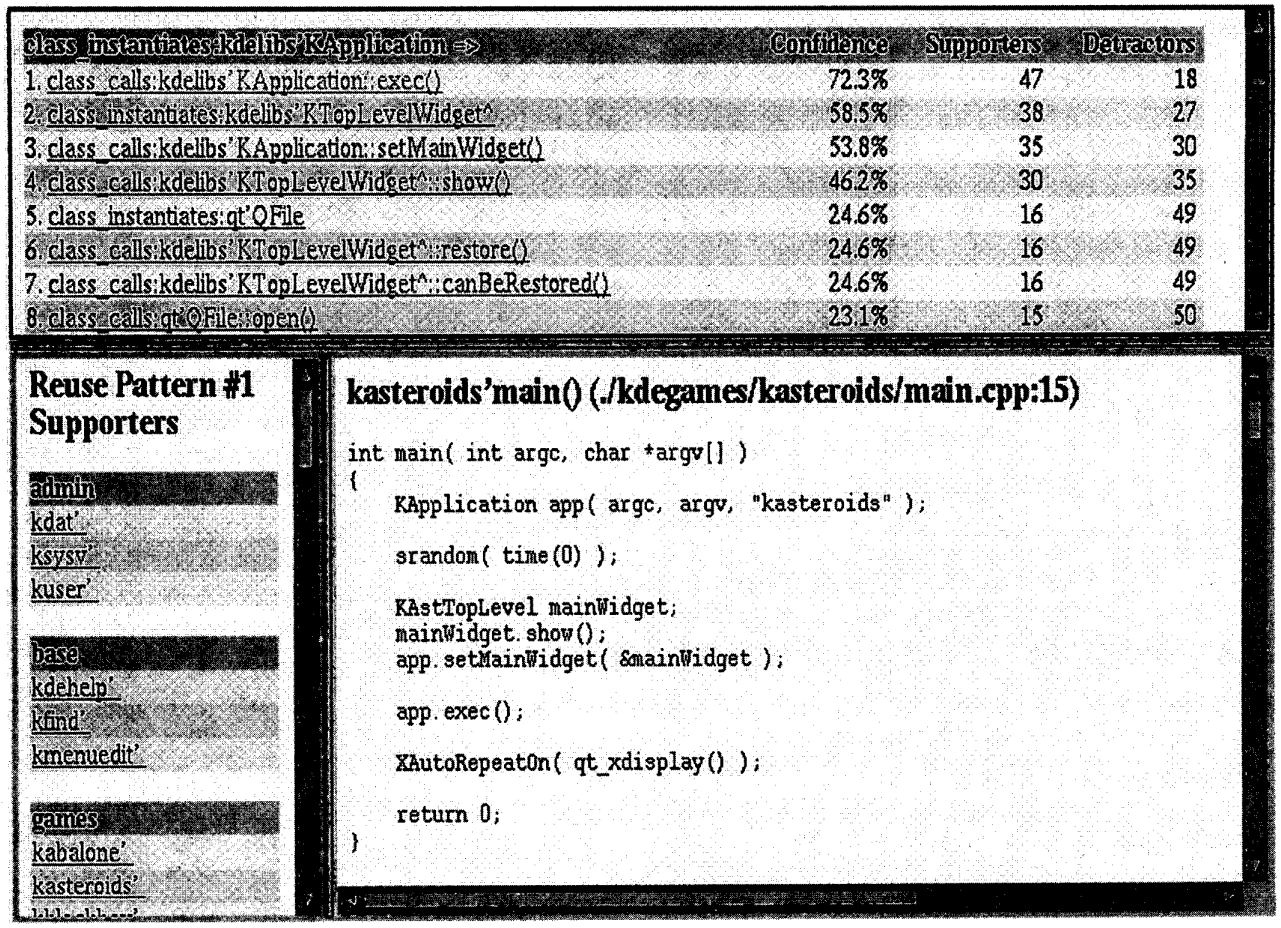
Exemplars

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 [6]. By browsing these classes as well as their static relationships and dynamic interactions, one can get a general understanding of how the framework works in a small example.

While an exemplar may be helpful, it is a pre-selected toy example that may not be representative of "real-life" applications. Moreover, exemplars place an extra burden on the developers of the software library. In contrast, our approach allows the user to browse reuse patterns and the corresponding supporter and detractor classes in real-life applications. Moreover, the tool is automated and works on any existing code.

Reengineering Libraries

Recently, research has been done on reengineering libraries by analyzing their usage in several existing applications [10]. This is done by constructing a lattice that provides insights



class_instantiates:kdelibs'KApplication =>	Confidence	Supporters	Detractors
1. class_calls:kdelibs'KApplication::exec()	72.3%	47	18
2. class_instantiates:kdelibs'KTopLevelWidget^	58.5%	38	27
3. class_calls:kdelibs'KApplication::setMainWidget()	53.8%	35	30
4. class_calls:kdelibs'KTopLevelWidget^::show()	46.2%	30	35
5. class_instantiates:qt'QFile	24.6%	16	49
6. class_calls:kdelibs'KTopLevelWidget^::restore()	24.6%	16	49
7. class_calls:kdelibs'KTopLevelWidget^::canBeRestored()	24.6%	16	49
8. class_calls:qt'QFile::open()	23.1%	15	50

Figure 3: KApplication reuse patterns. Clicking on kasteroids', a supporter of reuse pattern #1, yields the code on the right.

class_inherits:qt'QObject =>	Confidence	Supporters	Detractors
1. class_instantiates:qt'QFile	12.4%	17	120
2. class_calls:qt'QFile::open()	12.4%	17	120
3. class_calls:qt'QFile::close()	11.7%	16	121
4. class_calls:qt'QObject^::disconnect()	11.7%	16	121
class_instantiates:qt'QObject^ =>	**Confidence**	**Supporters**	**Detractors**
5. class_instantiates:qt'QMenuData^	13.5%	115	734
6. class_calls:qt'QMenuData^::insertItem()	13.5%	115	734
class_overrides:qt'QObject^::eventFilter() =>	**Confidence**	**Supporters**	**Detractors**
7. class_calls:qt'QEvent::type()	97.2%	35	1
8. class_calls:qt'QObject^::installEventFilter()	88.9%	32	4
9. class_calls:qt'QMouseEvent::button()	47.2%	17	19
10. class_instantiates:qt'QMenuData^	44.4%	16	20
11. class_calls:qt'QMenuData^::insertItem()	44.4%	16	20
class_calls:qt'QObject^::startTimer() =>	**Confidence**	**Supporters**	**Detractors**
12. class_overrides:qt'QObject^::timerEvent()	95.2%	40	2
13. class_calls:qt'QObject^::killTimers()	50.0%	21	21
class_calls:qt'QObject^::killTimers() =>	**Confidence**	**Supporters**	**Detractors**
14. class_calls:qt'QObject^::startTimer()	87.5%	21	3
15. class_overrides:qt'QObject^::timerEvent()	87.5%	21	3

Figure 4: QObject reuse patterns.

into the usage of the class hierarchy in a specific context. Such a lattice can be used to reengineer the library class hierarchy to better reflect standard usage. In contrast, we are interested in helping novice users learn to use a library to write new applications — not reengineer the library itself.

This different perspective has led us to: (1) initiate a new research direction for mining code for the purposes of illustrating characteristic code reuse; (2) use data mining techniques that scale to a hundred or more applications — not just a few examples for which confidence and support measures would not be meaningful; (3) look for different kinds of reuse patterns that are more helpful for demonstrating reuse of the library classes; and (4) construct a tool whose user interface is aimed at users of a software library rather than its developers.

Other Work involving Data Mining

Researchers have used data mining and related techniques for a variety of purposes. For example, data mining has been used to: discover likely program invariants [5]; infer specifications in software [3]; and decompose a software system into data cohesive subsystems to assist developers with reengineering and maintenance tasks [4]. The last of these is the only other work we are aware of that uses association rule mining in the software engineering domain.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how data mining can be used to discover library reuse patterns in existing applications. Specifically, we considered the problem of discovering library classes and member functions that are typically reused in combination by application classes.

This paper improves upon our earlier research using "association rules" [8] by taking into account the inheritance hierarchy using "generalized association rules". This has turned out to be non-trivial due to the significantly larger number of rules that arise as a result. Consequently, pruning is important and we showed several ways in which it can be done.

By browsing generalized association rules, a developer can discover patterns in library usage in a way that takes into account inheritance relationships. We have illustrated the approach using our tool, CodeWeb, by demonstrating characteristic ways in which applications reuse classes in the KDE application framework. We have observed that some important rules would not have been found without taking into account the inheritance hierarchy.

One can view our general approach to mining reuse patterns 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 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 and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference*, pages 487–499, 1994.

[2] M. Chen, J. Han, and P. S. Yu. Data mining: An overview from a database perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, 1996.

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

[4] 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.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

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

[7] N. Megiddo and R. Srikant. Discovering predictive assocation rules. In *Proceedings of the 4th International Conference on Knowledge Discovery in Databases and Data Mining*, 1998.

[8] A. Michail. Data mining library reuse patterns in user-selected applications. In *14th IEEE International Conference on Automated Software Engineering*, pages 24–33, 1999.

[9] G. Piatetsky-Shapiro and W. J. Frawley. *Knowledge Discovery in Databases*. AAAI/MIT Press, 1991.

[10] 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.

[11] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st Very Large Data Bases Conference*, 1995.

[12] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995.