CodeWeb: Data Mining Library Reuse Patterns

Amir Michail
School of Computer Science and Engineering
University of New South Wales
amichail@cse.unsw.edu.au

1 Introduction

While popular commercial libraries, such as MFC, are usually well-documented, open source or internally developed libraries are often not. In either case — and particularly in the latter — developers today learn to use a software library not just from its documentation but also from toy examples and existing real-life application code (e.g., by using grep or looking at browse information).

The CodeWeb tool [6, 7, 8] takes this simple idea further by a deeper analysis of a large collection of applications to see what characteristic usage of the library is like. Specifically, the tool uses data mining techniques to discover so-called "reuse patterns". Reuse patterns can be used to guide and check library usage.

Our tool-based approach alleviates problems with learning characteristic library usage in the following ways: (1) by using many real-life applications instead of a few toy programs, we can demonstrate reuse of many library classes in more numerous and deeper ways; (2) by using automated techniques, reuse patterns can always be kept up to date with respect to the most recent version of the library and applications; and (3) by leveraging existing applications and using data mining technology, we complement manually constructed tutorials and toy programs if they are available and provide an alternative source of information if they are not.

We demonstrate our approach by showing how the KDE core libraries are used in real-life KDE applications. Moreover, we look at a recently developed feature that helps software developers port an application from an old version of a library to a new one. Specifically, we consider porting KDE applications from version 1 of the KDE core libraries to version 2.

2 KDE

KDE is an open source project whose aim is to provide a powerful graphical desktop environment for UNIX workstations that rivals Microsoft Windows [11]. In what follows, we shall show the results of (1) data mining usage of the KDE 1 core libraries in 79 applications that come with the KDE 1 distribution; (2) data mining usage of the KDE 2 core libraries in 125 applications that come with the KDE 2 distribution; and (3) data mining differences in core library usage among KDE 1 and KDE 2 applications.

3 Reuse Table

A *reuse table* shows the percentage of existing applications that use various library classes. If a library class has been used by many applications in the past, then it is likely to be useful in future applications also — and so it is certainly worth knowing about. For example, we see from Figure 1, (a) that KApplication is used by 97.5% of all KDE 1 applications and 76.8% of KDE 2 applications. Consequently, we should certainly consider using this class in any new KDE application that we write.

The reuse table also shows two *usage deltas* for each library class. The first usage delta indicates the percentage of KDE 2 applications that use a particular class minus the percentage of KDE 1 applications that use that class. If the usage delta is a large positive number, then this indicates that there is a significant increase in the use of that class in KDE 2, so one may consider using this class when porting an application from KDE 1 to KDE 2. Similarly, if the usage delta is a large negative number, then one may consider not using the class any longer.

The second usage delta uses a different technique to provide similar — but possibly more accurate — information. (In future versions of the tool, both deltas may be

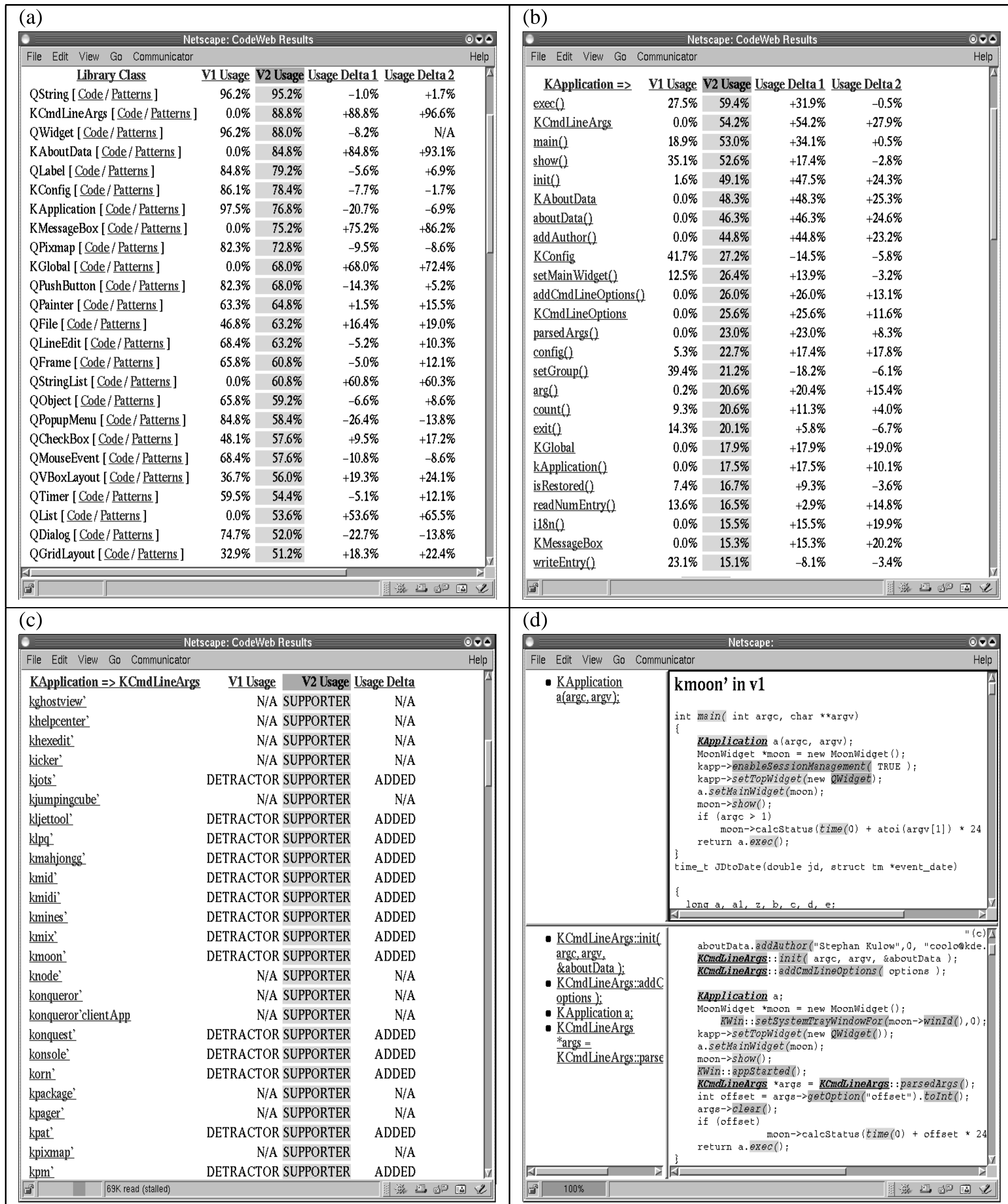

Figure 1: CodeWeb screenshots.

combined to produce one robust measure of change.) The second usage delta is calculated by considering: (1) the percentage of those *applications present in both versions* that use the library class in v2 but not in v1; and (2) the percentage of those *applications present in both versions* that use the library class in v1 but not in v2. The second percentage is shown in negated form (as it indicates that the library class is no longer used in v2).

In practice, we show only one percentage which is the larger of the absolute value of these two percentages. For example, if this delta yields $+25\%$ (to indicate adding a library class) and -50% (to indicate removing the class), then we only show the larger number -50% to indicate that the library class is removed in 50% of the applications ported from KDE 1 to KDE 2. Typically, one number is much larger than the other in absolute value, so there is no point in showing both.

From Figure 1, (a), we see that KCmdLineArgs has a usage delta 1 of +88.8% and a usage delta 2 of +96.6%. Indeed, this a new class introduced in KDE 2 to help with processing command line arguments by automatically taking into account KDE specific options. Many applications now use it in KDE 2, so we should consider using it in our applications also.

4 Reuse Patterns

Reuse patterns show characteristic ways in which library classes have been used in existing applications. (See Figure 1, (b).) Typically, one would first browse the reuse table to identify fundamental library classes of interest and then browse their reuse patterns to see how such classes are typically used in practice. By combining reuse patterns with the library reference (which we assume exists — at the very least as comments in the code), we start to approach the knowledge offered by tutorials.

Clicking on the "Patterns" link to the right of KApplication in Figure 1, (a) shows the reuse patterns in Figure 1, (b). Reuse patterns are basically if/then rules which indicate that application classes that contain the antecedent tend to also contain the consequent. For example, we see that application classes with KApplication in the code tend to also contain exec() and KCmdLineArgs. Rules of this form — which are known as *association rules* — are of extensive interest in the data mining community [1, 2, 3].

For each rule, four percentages are shown. The first two, under V1 and V2 usage, indicate the "strength" of the rule in KDE 1 and KDE 2. To be more precise, we first need to define the notion of *confidence* [1, 2], which is the percentage of application classes containing the antecedent that also contain the consequent.

The first two percentages in Figure 1, (b) show the *improvement*, which is defined as the confidence of a rule minus the percentage of application classes that contain the consequent of that rule [3]. Improvement is a better indicator of rule strength since confidence can be quite misleading — particularly if the presence of the antecedent actually *decreases* the likelihood of finding the consequent (in which case, the improvement will be negative).

We also show two *usage deltas* for each reuse pattern — analogous to the two in the reuse table. For example, Figure 1, (b) clearly shows the usage of new KDE 2 classes such as KCmdLineArgs and KAboutData, whenever KApplication is used, as is indicated by the large positive usage deltas.

The first usage delta is simply the improvement of the rule in KDE 2 minus the improvement of the rule in KDE 1. The second usage delta is calculated by considering: (1) the percentage of those *application classes present in both versions containing the antecedent in v1* in which the consequent is not present in v1 but is present in v2; and (2) the percentage of those *applications classes present in both versions containing the antecedent in v1* in which the consequent is present in v1 but is not present in v2. Again, we look at "improvement" by subtracting away the percentage of application classes present in both versions in which the corresponding consequent item is added/deleted (regardless of whether the antecedent is present in v1).

5 Application Source Code

The tool provides direct access to the application source code for examples of library usage. For example, clicking on KCmdLineArgs in Figure 1, (b) yields a list of application classes for the reuse pattern KApplication ⇒ KCmdLineArgs in Figure 1, (c). More specifically, the system shows a list of all application classes that contain the antecedent (e.g., KApplication in this case). Alongside each class, the tool indicates whether it supports the pattern (that is, it contains both the antecedent and consequent) or detracts from the pattern (that is, it contains the antecedent but not the consequent). Additionally, the usage delta column indicates whether the consequent is: (1) present in KDE 2 but not KDE 1 (that is, it was added); (2)

present in KDE 1 but not KDE 2 (that it, it was deleted); (3) present in both; or (4) present in neither.

Clicking on an application class in Figure 1, (c) shows the class source in KDE 1 and KDE 2 as demonstrated in Figure 1, (d) for the application class kmoon'. This is actually the "global" class for kmoon which contains all global functions and variables in that application. Other classes such as konqueror'clientApp denote a standard class (e.g., clientApp) in a particular application (e.g., konqueror).

6 Related Work

Much of the research on tool support for software reuse has focused on component retrieval systems whereby the user submits information about the required component — perhaps by specifying a few keywords — and the system returns the closest matches it finds in the software library [4, 9]. Yet, even if we ignore the technical challenges in such approaches (e.g., vocabulary mismatch), observe that this query-based paradigm is limited in two fundamental ways. First, the user may not be familiar with the domain and may not know what kind of components to look for in the library. This is particularly true with object-oriented frameworks where the inheritance hierarchy is deep and where lots of non-obvious choice is present. Second, even if the user does find the components of interest, they may not work together to accomplish the desired task. Even if the components can be made to work together to provide the desired functionality, the component retrieval system certainly doesn't tell the user how.

Other researchers have also observed the inadequacy of the component retrieval approach to software reuse. For example, there is other research that looks at typical library usage such as that on exemplars [5] and reengineering libraries [10]. While the first method addresses the same problem we do, it requires sophisticated domain analysis by an expert. The second method analyses library usage in existing applications in an automated manner, but the method is designed to help developers reengineer the inheritance hierarchy of a framework rather than help users develop applications using a framework. Moreover, our method scales better than the lattice clustering approach used in the second method and additionally supports porting applications from one version of the library to another.

References

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

[2] 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.

[3] R. J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 188–197, 1999.

[4] 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.

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

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

[7] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering*, 2000.

[8] A. Michail and D. Notkin. Illustrating object-oriented library reuse by example: A tool-based approach. In *13th IEEE International Conference on Automated Software Engineering*, 1998.

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

[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] KDE Team. What is KDE? http://www.kde.org/whatiskde/index.html.