Using Library Dependencies for Clustering

Jochen Quante
Software Engineering Group, FB03 Informatik, Universität Bremen
quante@informatik.uni-bremen.de

Abstract: Software clustering is an established approach to automatic architecture recovery. It groups components that are in some way similar to each other. Usually, the similarity measure is based on the dependencies between components of a system. Library dependencies are intentionally ignored during the clustering process – otherwise, system components would be clustered with library components they use.

We propose to particularly look at the dependencies on external components or libraries to learn more about an application's high-level structure. The number of dependencies of a component from different kinds of libraries provides insightful information about the component's functionality. Our case study illustrates the potential of this idea.

1 Introduction

Software clustering is a well-known technique that can help in the recovery process of a software's architecture. Clustering creates groups of similar elements ("clusters"), based on some similarity criterion. Each cluster then hopefully corresponds to some more abstract entity of the system. Many different similarity measures have been proposed for this purpose. They are usually based on internal system dependencies [Sch91, TH00], and their goal often is to maximize cohesion and minimize coupling – bearing in mind that this principle may have been followed during design. There are also approaches that exploit other sources of information such as naming [AL98, GKS97] or ownership information [BH98]. To our knowledge, all these approaches ignore library dependencies for clustering, although those could be a valuable source of information.

Libraries are collections of components[1] that solve certain recurring problems. Each of these problems is from a certain domain. Examples for such library domains include graphical user interfaces (GUI), database access, logging, XML parsing, communication, etc. Specially in the Java programming language, libraries are used quite intensively. The Java 6 runtime environment (JRE) alone contains about 16,000 classes, and there are thousands of 3rd party libraries available for many different purposes.

If a library is used by a system and if we know what that library does, we can detect where functionality that is related to that library is implemented by analyzing where it is used. For example, creation and management of user interfaces is usually done by using the API of some GUI library. When usages of such a library are encountered, we can conclude that

[1] *Component* in this paper means a software artifact at any level of granularity, e. g. module, method, etc..

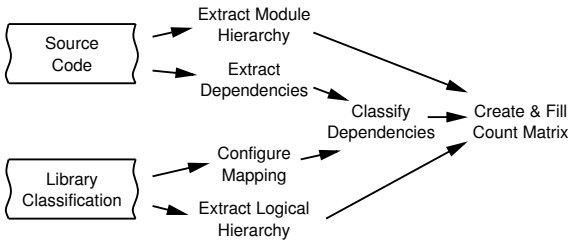

Figure 1: Method overview

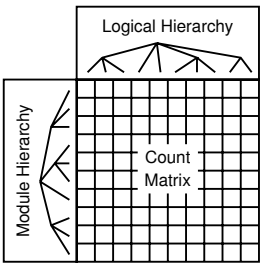

Figure 2: Hierarchical count matrix

the using component has something to do with the GUI. Similarly, database accesses, file accesses or other functionality that uses some library can be localized. Also, looking at the *number* of library dependencies should give additional hints about a component's role within an application.

We suggest that it should be a good idea to investigate library dependencies in order to gain an initial understanding of the functionality of the components of a given system that uses libraries. In the following sections, we will describe how these dependencies can be categorized, counted, visualized and used for clustering.

2 Method and Application Sketch

Data Collection. Figure 1 shows an overview of the process we propose. The analysis is based on the source code and knowledge about libraries, so first we need a library classification. The classification is used to decide what a concrete part of a library is used for. It is also used to construct a *logical hierarchy* that describes all available library functionality on an abstract level. Then, information about all modules and dependencies between them are extracted from the source code. From this information, the *module hierarchy* as defined by package or directory structure is extracted. These two hierarchies span the axes of a count matrix (see Figure 2). Each dependency is then classified, and the corresponding counter in the count matrix is increased by one.

Matrix Visualization. The resulting hierarchical matrix may become quite large, which impedes its immediate visualization. Fortunately, there are meaningful hierarchies on both axes, which allows us to start with collapsed hierarchies that can be expanded as needed to delve into details. This concept is known from Dependency Structure Matrices (DSM [SJSJ05]). Our library dependency counters are an ideal candidate for visualization as a DSM. They can be regarded as a bipartite graph: One partition contains the system's hierarchy as defined by the directory or package structure, while the other partition contains the logical hierarchy as defined by the mapping. Therefore, one axis of the DSM is spanned by the module hierarchy, and the other one contains only the logical hierarchy.

Clustering. Interactive exploration of usage counts can probably shed some light on certain components' meaning. However, it would be helpful to automatically detect groups

with a similar library usage profile, since this could indicate related functionality within the application. This group building is called clustering.

For clustering library usage profiles, we use *agglomerative hierarchical clustering*. This clustering algorithm works as follows. Initially, each feature vector is put into an own cluster. Then, the two clusters with minimum distance (according to some distance measure) are joined into one cluster. This step is repeated until either the minimum distance is above some threshold, or until a given number of desired clusters is reached.

In our case, a feature vector corresponds to one row in the count matrix. To gain meaningful results, it is important to assure the same granularity for each row. For example, we could choose to have only classes or only packages, but not a mixture of both. As the distance metric, we propose to use the following formula:

$$d(A,B) := \sqrt{\sum_{i=1}^{n} w_i \cdot (t(a_i) - t(b_i))^2}, \qquad t(x) := \log_{10}(x+1)$$

with $A = (a_1, \ldots, a_n)$, $B = (b_1, \ldots, b_n)$. The distance metric is based on the ordinary euclidean distance. The factors $w_i \in [0,1]$ are used to assign a weight to each node of the logical hierarchy. This is particularly useful for removing counters of logical entities that represent cross-cutting concerns like logging, or for reducing counters that do not have a meaning as strong as others. The function $t(x)$ is chosen in a way that $|t(x) - t(y)|$ is large when x and y are close to zero, but is small when x and y are greater numbers and close to each other. Values close to 0, which express whether there is a dependency or not, should have a large impact, while the difference between larger values of the same magnitude is rather small. By using the logarithm, only the order of magnitude of a counter is considered. The distance metric is extended to vectors by applying d on the average of all vectors that belong to each cluster.

3 Case Study

To evaluate if the proposed method of counting, visualizing and clustering library usages really delivers useful results, we apply it on a system called **cfgradebook**. This is a system for managing courses, students and grades. It is written in Java and comprises 28 KSLOC. The system uses JDO for database access, a Swing GUI, client/server communication through sockets, and implements an interface (*Mems*) to an external web application.

Figure 3 shows a screenshot of the count matrix **visualization** for cfgradebook. On the left side, the hierarchical structure of packages, classes and methods is displayed. On top of the matrix, the logical hierarchy is shown. The matrix itself is filled with the corresponding accumulated library usage counts. A lot of information about the system is provided by this matrix. For example, package `cfcommon` contains 3,909 static dependencies on database libraries, which indicates that most of the database code occurs within this package. Only package `cfserver.jdo` contains additional database code. Similarly, we can see that there is no GUI code at all in `cfcommon` and `cfserver` – only `cfclient` contains all the GUI code. We can also see that the *Mems* interface's implementation is distributed over all three base packages. By expanding the corresponding nodes, we could see exactly

Figure 3: Screenshot of count matrix visualization for cfgradebook. Different gray tones represent different depth in both hierarchies. Empty cells indicate a count value of zero.

No.	Cluster	#Cls
1	IO 2.4, Net 1.3	2
2	IO 2.1	6
3	Mems 1.9, Text 1.3, IO 0.6	1
4	GUI 1.4	47
5	DB 1.3, IO 1.0	35
6	Mems 1.3, Security 0.5	1
7	Net 1.3, Security 0.3	2
8	IO 1.2, Mems 0.8, Text 0.2	2
9	IO 0.6	26
10	Unclassified	77

Figure 4: Results of clustering cfgradebook for a maximum difference of 1.0 (Cls = Classes).

where it is used, as is in this screenshot visible for `cfserver`. When expanding the logical hierarchy of *Net*, the matrix tells us that *SSL* is used for communications and that *RMI* is *not* used. The *Basic* columns contain information about techniques like reflection, multithreading, and use of standard collections. The numbers here indicate that reflection is not used and that thread handling is mainly contained within `cfserver.control`.

In summary, the matrix visualization of cfgradebook tells us a lot about the system. The displayed information and the form of presentation can be very useful for getting an initial overview of the system and it's functionality. We will now examine if clustering works as well.

For **clustering** the library usage counters of cfgradebook, we regard them at class level only. Logging and system libraries such as `java.lang.*` are assigned weight zero. Classes that do not use any libraries are excluded from clustering and are directly placed into an own cluster. This is true for 77 of the 199 classes and leaves 122 classes to be clustered.

Table 4 shows the clustering results for a maximum difference threshold of 1.0, ordered by highest average library dependency counts. For example, 2.4 is the highest average count of all library categories in all clusters, so it is on top of the list. The largest cluster (4) contains 47 GUI classes. This cluster covers all the pure GUI classes in the system. As can be seen from the cluster names, GUI classes are never mixed with other libraries in this case, which indicates that the GUI has consistently been separated from other concerns. Similarly, cluster (5) resembles all classes that are related to database use – including the

complete data model. The other clusters are smaller, but more specialized, and thus also provide interesting information about the system.

In summary, the resulting clusters nicely group related functionality. Components for GUI and data model are isolated, and several specialized single-class components can be identified as well. However, there are also clusters like (9) that are not particularly meaningful, but their weakness is identified by low average usage counter logs. There are also a lot of unclassified classes about which we cannot say anything, except that they do not use any libraries.

4 Conclusion

Our case study shows that visualization of library dependency counts as a DSM can provide insightful information about a system. For example, it tells us if certain concerns are separated in the system's architecture. If this is the case, features that relate to libraries can be easily located. Clustering allows automatic evaluation of this matrix. The resulting clusters may correspond to conceptual components, and it may provide useful results where pure visualization of the matrix is not adequate.

The case study also shows that library dependency information alone is not sufficient for reconstructing conceptual components exactly. In particular, large portions of a system remain unclassified with this approach, and some clusters require further refinement. Therefore, we suggest to combine library dependency counting with traditional clustering techniques, for example techniques based on internal dependencies. This combination should result in improved clusterings, since both approaches individually deliver good results, and they exploit orthogonal information.

References

[AL98] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names: a new file clustering criterion. In *Proc. of 20th ICSE*, pages 84–93, 1998.

[BH98] Ivan T. Bowman and Richard C. Holt. Software Architecture Recovery Using Conway's Law. In *Proc. of the Centre for Advanced Studies Conference (CASCON '98)*, pages 123–133, 1998.

[GKS97] Jean-Francois Girard, Rainer Koschke, and Georg Schied. A Metric-based Approach to Detect Abstract Data Types and State Encapsulations. In *Proc. of the 1997 Conf. Automated Software Engineering (ASE)*, pages 82–89, 1997.

[Sch91] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. of 13th ICSE*, pages 83–92, 1991.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proc. of 20th OOPSLA*, pages 167–176, 2005.

[TH00] Vassilios Tzerpos and Richard C. Holt. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proc, of 7th WCRE*, pages 258–267, 2000.