An Analysis of Dependence on Third-party Libraries in Open Source and Proprietary Systems

Steven Raemaekers*†, Arie van Deursen† and Joost Visser*

* Software Improvement Group, Amsterdam, The Netherlands
E-mail {s.raemaekers,j.visser}@sig.eu
† Delft University of Technology, Delft, The Netherlands
E-mail {s.b.a.raemaekers, arie.vandeursen}@tudelft.nl

***Abstract*—The usage of third-party libraries can decrease development time and cost through reuse of existing pieces of functionality. However, little is known about the actual usage of third-party libraries in real-world applications. In this paper, we investigate the frequency of use of third-party libraries in a corpus of proprietary and open source systems. This information is used in a commonality rating, which provides information on the frequency of use of particular libraries and on the degree of dependence on third-party libraries in a software system. This information can be used in the decision to prefer a certain library over another or to estimate the amount of exposure to possible risks present in these libraries. In future work, this rating can be validated against different kinds of risk indicators.**

***Index Terms*—Third-party libraries; Risk management; API Usage; Software Reuse; Component-based development;**

I. Introduction

The usage of third-party libraries can save development time and effort by removing the need to rebuild already existing functionality [2], [7], [9]. According to a survey conducted by Forrester Consulting [1], software developers have embraced the use of third-party libraries: nearly all of 336 software companies in the survey work with some form of third-party software, and 40% of respondents even work with more than five third-party library suppliers. For a systematic review of the advantages of software reuse in general, see [8].

Except for download statistics, which are often available on the websites of these libraries, little is known about the actual frequency of use in proprietary and open source systems. We therefore perform this analysis in this paper and we propose a 'commonality' rating which captures the commonality of a third-party library in our corpus. The number of third-party libraries used in a system is also used in this rating to determine the scale of dependency on third-party libraries, and to estimate the exposure to possible risks present in these libraries.

This paper is structured as follows. First, related work will be discussed in section II. Definitions used in this paper can be found in section III. We will present our rating in section IV. We will present the frequencies and the result of our rating in section IV-D. Finally, we will present ways to validate this rating and describe an experiment to correlate this rating with specific risk indicators. This can be found in section VII.

This work was partly funded by the RAAK-PRO project EQuA (Early Quality Assurance in Software Production) of the Stichting Innovatie Alliantie.

II. Related work

Considerable research has been performed in the area of software reuse [4] and component-based development. The usage of third-party libraries is one form of software reuse, but the focus of software reuse literature is mostly on how to design a system in a way that makes reuse of components possible in general [10], [11]. It is not necessarily concerned with measurements in individual libraries, while our research focuses on gathering a specific indicator for a selected set of third-party libraries.

In the area of component-based development (CBD), research [3], [5] has been performed to help in identifying suitable components for reuse. Often, a standard set of metrics that is normally used to measure software quality in general is applied to components to be marked for reuse. For instance, Gui [5] focuses on coupling metrics to rank the reusability of software components. This paper introduces a new measurement which is not included in the standard set of software quality metrics.

The work of Lämmel [6] is perhaps most closely related to this study. Lämmel studies API usage in a large set of systems, but is only concerned with API usage in a set of open source systems from SourceForge[1]. Also, no way of rating an individual library is proposed to use this information.

III. Definitions

In this paper, we define a *third-party library* to be a reusable software component developed to be used by an entity other than the original developer of the component. Apache log4j and Apache Commons Collections are typical examples of third-party libraries. A third-party library can also be seen as a system which uses other third-party libraries by itself.

Furthermore, we make a distinction between *system-level packages*, *top-level packages* and *subpackages*. For instance, in `org.apache.tools.ant.task`, `org.apache.tools.ant` is a system-level package and `task` a top-level package. Sub-level packages are all packages one or multiple levels below top-level packages.

[1] http://www.sourceforge.net

IV. Rating based on frequency of use

Our proposed rating gives information for third-party libraries as well as for systems that use these libraries. To achieve this, frequencies of use are obtained through the extraction of import statements from Java files. Table I shows an overview of the systems included in our dataset.

A. Dataset

Our dataset consists of Java systems and libraries and is chosen to represent a wide range of business domains and functions. We use a set of 106 open source systems from the Qualitas Corpus [12] and a set of 178 proprietary systems of which source code is available at the Software Improvement Group (SIG).

Open source systems	min	p25	p50	p75	p95	max
KLOC	2	22	53	113	438	2283
# files	32	201	468	1079	2570	20457
# 3rd party libs	0	5	9	25	73	232

Proprietary systems	min	p25	p50	p75	p95	max
KLOC	0.7	28	84	193	1140	4752
# files	6	203	652	1256	4042	9597
# 3rd party libs	0	7	11	23	36	52

TABLE I
DESCRIPTIVE STATISTICS OF INCLUDED SYSTEMS

The Qualitas Corpus is a curated collection of selected systems, which increases the replicability and comparability of our study to other studies using the same corpus. The collection of systems available at the SIG gives invaluable insight in the usage of third-party libraries in proprietary systems.

Java was chosen as a language because a large number of open source and proprietary systems have been written in it. We also expect that Java systems are representative for systems written in other object-oriented languages.

B. Obtaining frequency of use

In order to obtain the frequency of use of third-party libraries, import and package statements are extracted from our set of systems. All files with a **.java* extension are scanned for import statements (`import <package_name>;`) and package declarations (`package <package_name>;`). Java files have only one package declaration per file and can have multiple import declarations per file. For confidentiality reasons, names of commercial systems cannot be provided in this paper.

The result of this extraction is an unfiltered list of import statements and package declarations, which are reduced to a list of libraries as follows. If a package declaration is defined in a file (trailing class name and trailing `.*` are removed) and that same package is imported in another file in the same system, we consider the import to be internal. All other imports are considered to be external. To reduce the number of imports to system-level packages we look through the list of imports per system manually (`org.apache.maven.task` and `org.apache.maven.plugin` result in one third-party library dependency: `org.apache.maven`). From this process, frequencies of imports of libraries are obtained by aggregating import counts for all files per system. For each system, each library that is imported is counted only once.

We only analyze library dependencies in non-test code since these dependencies are needed when the code is actually being used. Therefore, test code is removed from our analysis. Also, the number of references to libraries inside test classes is negligible In the analysis of import statements, references to libraries starting with `java.`, `javax.` and `sun.` are ignored since we do not classify the base class libraries as third-party.

C. Calculating a rating based on frequency of use

After extraction and processing of import and package statements, a rating is calculated both for third-party libraries and systems that use these libraries. Ideally, a rating should be easy to obtain and should have a desirable statistical distribution (systems can be discriminated with this metric). Our rating satisfies these properties since it is relatively simple to calculate, and given our results in the next section, systems can be discriminated with it as well.

The rating for a specific *library* that we propose in this paper is the number of different systems it is used in divided by the total number of systems in our dataset. The rating for a *system* is the sum of all ratings of the libraries it contains, divided by the square of the number of libraries.

The rationale behind this rating is as follows. We assume that when a library is used frequently in different projects, there must have been a reason to do so: a large number of different project teams apparently considers it safe enough to use and make a rational decision to prefer a certain library over another. We expect that people are risk-averse in their choice of third-party libraries, and that therefore people tend to prefer safer libraries over less safer ones. We make use of the collective judgment of these development teams in our rating. We expect that our rating correlates with different kinds of risks present in these libraries, but validation of this hypothesis will not be performed in this paper. Section VII explains how this could be tested experimentally.

We also assume that the more third-party library dependencies a system has, the higher the exposure to risk in these libraries becomes. Therefore we divide the average rating of a system by the number of dependencies per system, thus punishing a large number of third-party libraries.

Formally, the ratings are defined as follows. Let P be the set of all systems under analysis and m be the size of P. Each $S \in P$ uses a set L_S of third-party libraries. The rating for a given library L is given by $R(L)$ which is defined as $R(L) = \frac{f(L)}{m}$ where $f(L)$ is the frequency of use of L in all the systems of P. The rating of a given system S (Commonality Rating) is given by $CR(S)$ which is defined as

$$CR(S) = \frac{\sum_{i=1}^{|L_S|} R(L_i)}{|L_S|^2}$$

The ratings as calculated by these formulas are dimensionless and always range from 0 to 1. When a system does not

have any third-party library dependencies, the rating is 1. Two systems can be compared by their ratings, the system or library with the highest rating is preferred.

D. Results

The top 10 of frequently used third-party libraries can be found in Table II. A selection of open source systems with ratings based on the number of imported libraries can be seen in Table III. The first table shows the top 10 imports for the Qualitas Corpus, the second table for our set of proprietary systems.

Open source systems				
	Library name	*Description*	*#*	*%*
1	org.apache.tools.ant	Build tool	31	30.1%
2	org.apache.commons.logging	Logging framework	29	28.2%
3	org.apache.log4j	Logging framework	25	24.3%
4	org.apache.commons.collections	Collection extensions	18	17.5%
5	org.apache.commons.httpclient	HTTP client-side library	17	16.5%
6	org.apache.commons.lang	SDK base class extensions	15	14.6%
7	org.apache.xml	XML framework	15	14.6%
8	org.apache.commons.beanutils	JavaBeans utility classes	14	13.6%
9	org.apache.commons.codec	Encoder/decoder collection	14	13.6%
10	org.dom4j	XML processing library	14	13.6%

Proprietary systems				
	Library name	*Description*	*#*	*%*
1	org.apache.log4j	Logging framework	80	45.0%
2	org.apache.commons.lang	SDK base class extensions	68	38.2%
3	org.springframework	Dependency injection	58	32.6%
4	org.hibernate	Persistence framework	37	20.8%
5	org.apache.commons.beanutils	JavaBeans utility classes	30	16.9%
6	org.apache.commons.collections	Collection extensions	30	16.9%
7	org.apache.commons.logging	Logging framework	29	16.3%
8	org.joda.time	Date & time API	29	16.3%
9	org.apache.commons.io	IO utility library	20	11.2%
10	org.apache.xmlbeans	XML binding framework	20	11.2%

TABLE II
THE TOP 10 OF MOST IMPORTED THIRD-PARTY LIBRARIES

As can be seen in table II, there are slight differences between the two datasets. In the case of the Qualitas Corpus, the most frequently imported library is `org.apache.tools.ant`, and for proprietary systems `org.apache.log4j`. The Apache libraries are popular in both datasets, occupying 15 out of 20 places in the list.

The Spring framework and Hibernate only appear in the top 10 of commercial systems and not in the top 10 of the Qualitas Corpus. For the Spring framework, this is not surprising since this framework is especially developed for industrial systems. Table II also shows that 45.0% of proprietary systems use the number one import `org.apache.log4j`, while 30.0% of open source systems use the number one import `org.apache.tools.ant`. This is also true for most of the other libraries in the top 10 of proprietary systems; apparently there is more uniformity in used libraries in proprietary systems than in open source systems.

The values of the rating formula of a selection of systems from the Qualitas Corpus are shown in Table III. The number in parentheses behind each library is the number of times that library is used in the Qualitas Corpus. The number behind each system (prefixed with a p) is the percentile of the score of that system. For instance, only 2% of systems have a score higher than Quilt (p98). This also means that our rating is not distributed evenly from 0 to 1 but most systems have a value lower than 0.5. The ratings of other systems in the Qualitas Corpus can be found in the addendum.

System/Import	*Rating*
ProGuard 4.5.1	0.2925 (p100)
org.apache.tools.ant	0.2925 (31)
Quilt 0.6	0.0967 (p98)
org.apache.bcel	0.0755 (8)
org.apache.tools.ant	0.3113 (31)
Pooka 3.0	0.0283 (p81)
org.htmlparser	0.0283 (3)
PicoContainer 2.10.2	0.0094 (p60)
com.thoughtworks.paranamer	0.0094 (1)
Art of Illusion 2.8.1	0.0019 (p10)
buoy.widget	0.0094 (1)
com.jstatcom.component	0.0094 (1)
net.sourceforge.helpgui	0.0094 (1)
nik777.xlate	0.0094 (1)
org.jibble.pircbot	0.0094 (1)

TABLE III
A SELECTION OF RATINGS FOR SYSTEMS FROM THE QUALITAS CORPUS

As can be seen in the table, ProGuard, a Java class file shrinker and optimizer, has one third-party dependency, namely `org.apache.tools.ant`. This library is imported in 31 other systems in the Qualitas Corpus, and therefore this system receives a rating of $\frac{\frac{31}{106}}{1^2} = 0.2925$. Art of Illusion, a 3D modelling and ray tracing program, uses 5 relatively uncommon libraries which are only imported in this system, and therefore receives the lowest rating of the systems in this table: $\frac{5*\frac{1}{106}}{5^2} = 0.0019$.

V. DISCUSSION

Our analysis shows that frequency of use and the number of libraries used can give valuable insight in the usage of third-party libraries in a system. The same analysis can be performed by any organization which has disposal over a large enough set of systems.

A. Meaning of rating

In this paper, we created a rating which rates more common third-party libraries higher than less common ones, and systems with a large number of third-party dependencies get rated lower than systems with less third-party dependencies. This is based on the assumption that the more third-party libraries a system includes, the more 'exposed' a system is to risks present in these libraries. Risks that can be present in a third-party library are, for instance, the crash of the system when encountering a bug in a library, or the risk that the third-party library is not maintained by its original maintainers any more after which it may have to be replaced.

As explained before, our hypothesis is that commonly used libraries are more 'safe' with regards to these risks. We expect that the choice of a large group of people will reflect the amount of different kinds of risk present in these libraries. When there are two comparable libraries available and one is considered to be more safe to use, we expect that this will be reflected in the usage statistics of these libraries. For a possible validation of this hypothesis, see section VII.

B. Rare and useful

The approach described in this paper should considered to be a start to the assessment of commonality of third-party libraries, and should not be applied without contextual interpretation. For instance, a library with a higher rating is generally preferred over a library with a lower one, but when a library performs a very specific task and there are no alternatives, there is no choice but to use this library. This is a contextual fact that has to be taken into account. Generally speaking, it would be wise to prefer common libraries over more uncommon ones, provided that they implement the same functionality. To determine the predictive value of this rating, additional validation is required.

C. Cascading dependencies

It is possible that a third-party library contains references to other third-party libraries. We ignored these 'cascading dependencies' in our analysis, but should ideally be included since there is no difference between direct and indirect dependencies from a technical viewpoint. We did not include code of third party libraries and therefore import statements in these libraries were not collected. Source code of these libraries could also be included in future work to investigate cascading third party library dependencies.

VI. Threats to validity

A. Internal validity

To gather data for our analysis, manual inspection is required when reducing import statements to system-level packages. This is not an error-free process, but nevertheless necessary since we do not have another way to distinguish between system-level, top-level and sublevel packages (`org.apache.maven.plugin` and `org.apache.maven.model` can be reduced to `org.apache.maven`, but `org.apache.maven` and `org.apache.lucene` cannot be reduced to `org.apache`). A way to eliminate this threat would be to automatically determine whether a dependency is third-party by looking it up in a database of known third-party libraries or to apply more advanced heuristics.

B. External validity

Our choice of including only Java systems in our analysis has lead to a bias towards this language and may raise questions about the external validity of our study. To increase the external validity, the same experiment would have to be repeated with systems in other languages, which is possible for any language that supports importing packages or modules in a source file.

VII. Validation

To validate our hypothesis, the following experiment would have to be performed. First, for each library and for each specific risk, different indicators have to be collected, like the number of open bugs in issue tracking systems or the number of commits to the source code of the library. Systems need to be ranked based on each of these risks, so that the system with the highest risk is on the first place, and the system with the lowest risk is on the last place. A ranking based on our commonality rating also needs to be calculated. Each of these rankings can be compared to the rank of our commonality rating to see if there exists a correlation, for instance with Kendell's coefficient of concordance. If there turns out to be a correlation with, for instance, the number of open bugs in these libraries, then the commonality rating can be considered an easy-to-measure 'proxy' indicator for this risk.

VIII. Conclusion

In this paper, we presented an easy to calculate rating for the commonality of third-party libraries and the usage of these libraries in software systems. The contributions of this paper are:

1) An empirical study on the popularity of third-party libraries in open source and proprietary software systems;
2) A mechanism for rating libraries based on frequency of use in open source and proprietary systems;
3) A mechanism for rating systems in terms of their dependencies on (common or rare) third-party libraries;
4) A proposal for validation of our commonality rating as an indicator for risks present in these libraries.

References

[1] Software integrity risk report. Technical report, Forrester Consulting, 2011.

[2] M. T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio. An industrial case study on reuse oriented development. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 283–292, Washington, DC, USA, 2005. IEEE Computer Society.

[3] M. A. S. Boxall and S. Araban. Interface metrics for reusability analysis of components. In *Proceedings of the 2004 Australian Software Engineering Conference*, ASWEC '04, pages 40–51, Washington, DC, USA, 2004. IEEE Computer Society.

[4] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31:529–536, July 2005.

[5] G. Gui and P. D. Scott. Ranking reusability of software components using coupling metrics. *J. Syst. Softw.*, 80:1450–1459, September 2007.

[6] R. Lämmel, E. Pek, and J. Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[7] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*, 11:23–30, September 1994.

[8] P. Mohagheghi and R. Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Softw. Engg.*, 12:471–516, October 2007.

[9] M. Morisio, D. Romano, and I. Stamelos. Quality, productivity, and learning in framework-based development: An exploratory case study. *IEEE Transactions on Software Engineering*, 28:876–888, 2002.

[10] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Acm Press, 1997.

[12] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.