Measuring Software Library Stability Through Historical Version Analysis

Steven Raemaekers*†, Arie van Deursen† and Joost Visser*
*Software Improvement Group, Amsterdam, The Netherlands
Email: {s.raemaekers, j.visser}@sig.eu
†Software Engineering Research Group, TU Delft, The Netherlands
Email: arie.vandeursen@tudelft.nl

***Abstract*—Backward compatibility is a major concern for any library developer. In this paper, we evaluate how stable a set of frequently used third-party libraries is in terms of method removals, implementation change, the ratio of change in old methods to change in new ones and the percentage of new methods in each snapshot. We provide a motivating example of a commercial company which demonstrates several issues associated with the usage of third-party libraries. To obtain dependencies from software systems we developed a framework which extracts dependencies from Maven build files and which analyzes system and library code. We propose four metrics which provide different insights in the implementation and interface stability of a library. The usage frequency of library methods is utilized as a weight in the final metric and is obtained from a dataset of more than 2300 snapshots of 140 industrial Java systems. We finally describe three scenarios and an example of the application of our metrics.**

***Keywords*-Third-party Libraries; API Usage; API Stability; Software Reuse;**

I. Introduction

Backward compatibility is a major concern for any library developer. If a new version of a library introduces breaking changes, then system developers are either forced to update their system to work with the new version or they must keep using the old version of the library (for a visual example, see Figure 1). Library developers, on the other hand, want to release new versions of their software to include new features, improve existing ones or fix bugs. Library developers are constantly faced with a trade-off between keeping backward compatibility and live with mistakes from the past or start over and introduce breaking changes, but at the expense of a loss of backward compatibility. A good library should ideally be built in such way that the public interface is never broken, but this may prove to be impossible in practice.

An example of a company that spends great effort to keep their Application Programming Interface (API) backward compatible is Microsoft with its Windows SDK, which is used by all Windows programmers. Today, many old software systems still run on the latest version of Windows, thanks to a high degree of backward compatibility. If mistakes in the design of Windows were made in the past, these cannot be easily removed and would still be visible today.

Complete API stability may be hard to achieve, and in practice, different libraries can have different degrees of API stability. There are several different properties of libraries which could indicate this stability. For instance, if the number of parameters of a library method changes, library users have no choice but to change each place where a call to this method occurs. When a public method is removed in a next version of a library, developers are also required to remove all calls to this method. More subtle are internal implementation changes while method interfaces are being kept constant, but even those changes could have an impact on systems using these libraries, since behavior could change in an undesired way.

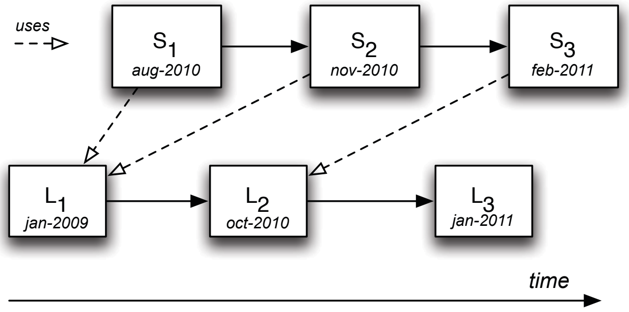

Figure 1. An example of the relation between different system and library versions. In this example, S_1 denotes the first version of the system, made in August 2010. This version uses library L_1 with date January 2009. The next version of the system, S_2, still uses library version L_1 while there is already a new library available, released in October 2010. This illustrates a possible delay in the adaptation of the latest library versions.

The goal of this paper is to introduce a way to measure interface and implementation stability. To that end we analyze historical values of metrics, weighted by the times methods, classes or packages are being used. To collect data on dependencies used in industrial systems we create an infrastructure to extract third-party library dependencies for Java as defined in Maven build files. We apply our metrics to the most frequently used Apache Commons libraries and we give an example of the application of our metrics to a library, for instance to determine if a library is in a state of maintenance or active development.

In Section II, we start with a motivating example of third-party library usage in a commercial company. In Section III, the problem statement and a definition of library stability are given. In Section IV, we discuss related work in the field of API usage, migration and evolution. In Section V, we describe our dataset used to calculate our library

978-1-4673-2312-3/12/$31.00 © 2012 IEEE

stability metrics. Section VI contains a description of our data collection and manipulation framework. In Section VII we describe variables, unit weights and historical weighting schemes. In Section VIII we present our metrics. Results are presented in Section IX. Possible scenarios and an example in which our metrics can be used are described in Section X. Finally, we discuss our work, threats to validity and future research directions in Sections XI and XII.

II. Motivating example

To describe the issues regarding dependencies on third-party libraries in large commercial and open source software systems, we give an example of a commercial company which uses several third-party libraries in a custom-developed web application of considerable size (approximately 4000 Java files and 200,000 lines of code). This application depends heavily on several libraries such as Spring, Apache Struts and Hibernate. For confidentiality reasons, the system and company name cannot be provided.

When the application was first built in 2004, third-party library dependencies were managed using Maven.[1] Version numbers of the latest versions which were available at that time were hard-coded in the configuration files of the project. These libraries were not updated to more recent version in the next 7 years, which resulted in a large "maintenance debt" of lagging versions. For instance, version 1.0 of the Acegi authentication and security framework (started in late 2003) was being used while this library was included in the Spring framework and was renamed "Spring Security" 2.0.0 in 2008. In the meantime, several breaking changes were introduced in new versions of the Spring Security framework as well as critical safety-related bug fixes and improvements.

Due to expected compatibility issues when upgrading Acegi, this update was deferred as long as possible. In the old setup, user authentication was handled through the Acegi library which communicated with an LDAP authentication server. To improve authentication and to facilitate single sign-on, Atlassian Crowd was contracted, a web-based authentication service. However, the Acegi framework was not capable of communicating with Atlassian Crowd and Acegi therefore had to be replaced. Spring Security was chosen as a natural successor although the library was rewritten from scratch and several breaking changes were introduced. The latest version available during the update process was 3.0.6, which changed significantly from Acegi 1.0.

Since Spring Security 3.0.6 is part of the more general Spring Framework, the entire Spring Framework had to be updated too. Considering the large dependence of the application on this framework, this would mean that a large part of the application had to be adapted to work with this new version. Since the Struts framework was also used and the new version of the Spring framework could not work with the old version of Struts, this had to be upgraded as well. Java code had to be adapted due to the transformation from Acegi to Spring Security. Also, since the syntax of the expression language used in Java Server Pages (JSP) was changed between Struts 2.0.9 and 2.2.3.1, all web pages in which dynamic content was presented using JSP had to be updated with the new syntax.

Eventually, a week was spent to implement the changes and upgrades. There was a test suite available, both in the form of unit tests written in Java and automated browser interface tests created with Selenium. Developers working on the system commented that without this test suite the impact of such an update would be much harder to assess.

This case illustrates several issues with third-party library dependencies. First, it shows the accumulation of maintenance debt when deferring updates of libraries. Second, it shows that there may come a moment in the future in which there is no choice but to update to a new version in which case a much larger effort has to be put in than in the case of smaller incremental updates. Third, it shows a case of a library that disregards backward compatibility and introduces breaking changes: the Struts library changed JSP expression language syntax which would require a large rework effort in systems using this syntax. Fourth, it also shows that transitive dependencies of included libraries can increase the total amount of work required to update to a new version of a library, even if an upgrade of these transitive dependencies was originally not intended. Finally, it shows the risk of using deprecated and legacy versions of libraries which can contain security weaknesses or critical bugs.

III. Problem statement

The example in the previous section shows that there are several issues involved with the usage of third-party libraries in general. Not every library is maintained with the notion of backward compatibility in mind, while any developer working with a third-party library immediately notices any change made to its public interface. The less library developers ensure continuity and stability of its public interface, the harder upgrading to the latest version of a library becomes. This poses a challenge for developers of an API, who have to think carefully about the public parts of an interface they publish. After releasing an API it is very difficult to make large-scale changes to it since other developers count on already released parts, and changing only even a small part of this interface will require developers to adjust their implementations. Since requirements keep changing and systems keep evolving over time, designing the correct interface that is stable and backward compatible enough in subsequent releases but also flexible enough to adapt to changing requirements can be challenging.

We do not address all of these issues in this paper, but we do provide a method to measure the *stability* of a

[1] See the online addendum at http://www.sig.eu/en/stability for URL references.

library, which could cause problems as mentioned above. We therefore introduce four metrics which provide insight on different aspects of implementation and interface stability. We consider an API to be stable if functionality is not removed from a public interface once it has been added. In the case of a Java system, this means that methods, classes or packages are not removed from the interface once they have been added and method signatures are not changed (adding/removing/changing parameters or renaming methods). We consider the implementation of a library to be stable with regard to a certain metric if metric values in a system are relatively constant through time.

The definition of library stability that we assume in this paper is the following: *library (in)stability is the degree to which the public interface or implementation of a software library changes through time in such way that it potentially requires users of this library to rework their implementations due to these changes*.

IV. Related Work

The problem of changing API interfaces has been recognized by other authors [1], [2], [4] and has been researched through different approaches than the approach of this paper. For instance, tools have been proposed to detect API evolution and to suggest refactorings to get up-to-date with the latest version of an API [4], [7]. Dagenais and Robillard [1] propose SemDiff, a tool that recommends replacements for framework methods that were accessed by a client program and deleted during the evolution of the framework.

Uddin et al [10] proposes a way to detect changes in the *usage* of an API, which differs from our work, in which we investigate changes over time in libraries themselves. The starting point of their data collection framework is similar to ours: they store client evolution patterns which represent a time series of changes which consists of added and removed methods calls to the API, with each snapshot having a time stamp. From this point on, however, Uddin et al take a different approach and investigate how to represent these client-side change patterns mathematically and how to infer temporal API usage patterns over time.

Other work in API usage mining often focuses on the detection of usage patterns of API methods in systems that call these methods. It can reveal how an API method is normally called and what preparing statements have been executed [11], [12]. Similarly, Thummalapenta [9] presents a framework for the detection of hotspots and coldspots in API's which can help to show relevant code examples in documentation of third-party libraries.

Furthermore, usage of certain libraries and specific methods has also been investigated. This research often shows usage statistics of certain parts of libraries ("hidden" or "public") [3], [5], [6] or collects statistics on the frequencies of use for methods in the library. In our work we want to use this type of information to help in the decision to include a certain library in a software project, or to choose a better alternative if one is available.

To our knowledge, taking into account the historical evolution to create metrics for library stability has not been done before. We believe this is valuable since it is a measure for the amount of backward compatibility of a third-party library, a property that could be of great interest to developers using this library since we expect it to be indicative for the amount of work required to update to a new version of a library.

V. Dataset

We present our approach in the context of Java systems. In particular, we assume the use of Maven, as this makes it possible to detect which library versions are used by a particular system. Java is appropriate to investigate as a programming language since 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. The central artifact repositories used by Maven facilitate large-scale downloading and analysis of dependencies.

Furthermore, our research focuses on one particular set of libraries: the Apache Commons libraries. In earlier work we have identified this as one of the most commonly used libraries [8] in Java systems, making it a suitable learning example. Some descriptive characteristics of the Apache Commons libraries are provided in Table I.

Since we are interested in the implications of stability on the actual *use* of a library, we need a set of systems making use of libraries. To that end, we use a collection of 2487 snapshots of 140 industrial Maven-based systems of which source code is available at the Software Improvement Group (SIG). The systems come from the same set we have used in earlier work [8]. Statistics on the use of the Apache Commons libraries by our set of subject systems are provided in Table II. In Table II, library versions that exist but are never included are omitted.

Library name	*LOC*	*Classes*	*#Mtds*	*#S*	*Latest*
Apache Commons Collections	26323	422	3945	6	dec-'11
Apache Commons Lang	19475	122	2338	6	jan-'11
Apache Commons HTTPClient	17171	171	1944	3	aug-'07
Apache Commons Beanutils	11375	127	1284	5	mar-'10
Apache Commons IO	8086	100	1053	7	oct-'11
Apache Commons Codec	4554	64	503	4	nov-'11
Apache Commons Logging	2680	27	311	3	nov-'07

Table I
Descriptive statistics of the Apache Commons libraries.
#Mtds=Nr. of Methods, #S=Nr. of Snapshots

VI. Analyzing Maven Dependencies

To obtain usage frequencies, Maven build files are scanned for third-party library dependencies. Each Maven project (`pom.xml`) file can contain a dependency section with information on the name and version of used libraries. For an example of a Maven dependency, see Figure 2. Our

Library	*Version*	*Date*	*Times used*
Commons Beanutils	1.6.1	18-feb-‘03	253
	1.7.0	02-aug-‘04	1776
	1.8.0	01-sep-‘08	86
	1.8.2	13-nov-‘09	61
	1.8.3	28-mar-‘10	67
Commons Codec	1.3	10-jul-‘04	684
	1.4	09-aug-‘09	622
	1.5	29-mar-‘11	129
	1.6	20-nov-‘11	1
Commons Collections	2.1	21-oct-‘02	317
	2.1.1	29-may-‘04	98
	3.0	25-jan-‘04	124
	3.1	23-jun-‘04	609
	3.2	14-may-‘06	1863
	3.2.1	07-dec-‘11	1375
Commons Httpclient	2.0.2	10-oct-‘04	237
	3.0.1	07-may-‘06	245
	3.1	18-aug-‘07	1053
Commons IO	1.1	10-oct-‘05	308
	1.2	19-mar-‘06	425
	1.3.1	13-feb-‘07	59
	1.3.2	02-jul-‘07	73
	1.4	21-jan-‘08	1076
	2.0.1	26-dec-‘10	188
	2.1	11-oct-‘11	27
Commons Lang	2.1	12-jun-‘05	1205
	2.2	28-jul-‘07	369
	2.3	13-feb-‘07	1475
	2.4	19-mar-‘08	1922
	2.5	07-apr-‘10	783
	2.6	16-jan-‘11	371
Commons Logging	1.0.4	10-jun-‘04	1507
	1.1	03-jun-‘07	2209
	1.1.1	22-nov-‘07	1563

Table II
THE USAGE STATISTICS OF THE APACHE COMMONS LIBRARIES

dataset contains multiple snapshots of systems on different points in time, each pointing to possibly different versions of third-party libraries. Each system and third-party library version is tagged with a version number and snapshot date.

Maven has a complex system to link the right versions of libraries to a deliverable, such as a jar file. This is not a trivial process since projects can contain multiple `pom.xml` files which can each point to different versions of the same library. Maven assures that only one version of a library is included inside a single deliverable.

```
<dependencies>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.1</version>
  </dependency>
</dependencies>
```

Figure 2. An example of a dependency in a Maven build file.

Dependencies can be analyzed and viewed in an hierarchical representation using the Maven dependency plugin but this requires a fully compiling project. This in turn requires the complete set of all `pom.xml` files that are referenced in a project. In our dataset, the collection of pom files is often incomplete and therefore a reconstruction has to be made which uses approximately the same rules as the resolving engine of Maven but can handle missing parent poms or otherwise incomplete references.

The Maven build file can contain dependency sections which contain a groupId, artifactId and version element (see Figure 2). The groupId, artifactId and version of a dependency together uniquely identify a certain library, which can be obtained through a public Maven repository. Specifying dependency versions was not required in older versions of Maven (before version 3) and versions of libraries are therefore often missing in pom files used with older versions of Maven. Also, a version can be mentioned in a "dependency management" section in a pom file higher in the directory hierarchy which specifies that a specific version of a library should be used, if it would ever be included.

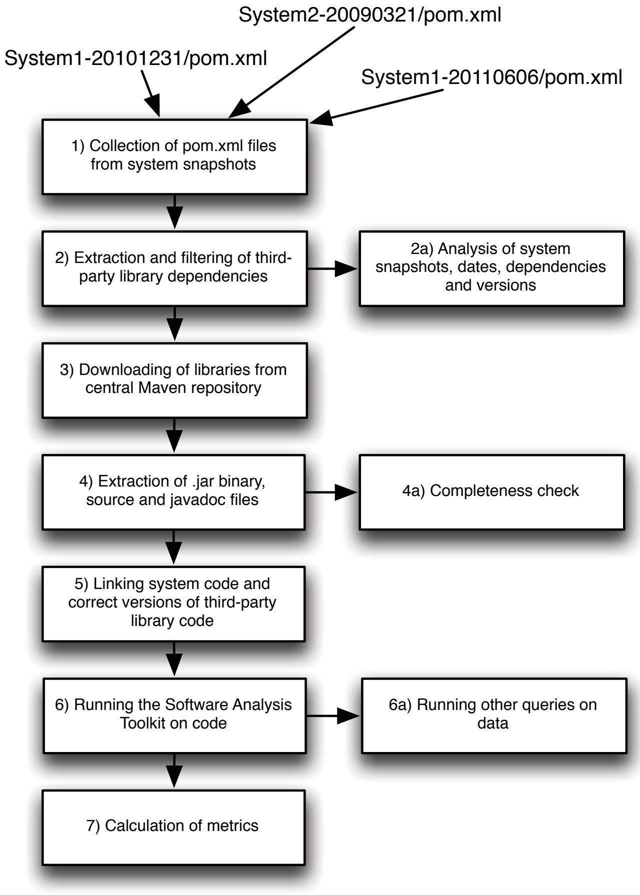
System2-20090321/pom.xml
System1-20101231/pom.xml
System1-20110606/pom.xml
1) Collection of pom.xml files from system snapshots
2) Extraction and filtering of third-party library dependencies
2a) Analysis of system snapshots, dates, dependencies and versions
3) Downloading of libraries from central Maven repository
4) Extraction of .jar binary, source and javadoc files
4a) Completeness check
5) Linking system code and correct versions of third-party library code
6) Running the Software Analysis Toolkit on code
6a) Running other queries on data
7) Calculation of metrics

Figure 3. The process of Maven repository mining. Starting with separate dependency files, the result is a database with all metrics per snapshot including source code of dependencies.

Pom files are ordered hierarchically, in such way that a pom file present in a child directory overrules settings from a pom file present in the parent folder of that directory. Pom files can specify module dependencies in which only a selection of poms can be included in a particular build target. Pom inheritance and module specification are two mechanisms which work independently of each other.

To reduce the complexity in resolving the right version number for each dependency, we make simplifying assumptions, which result in the selection of the latest version of a library when two or more different versions of the same library are included. For a complete list of assumptions, see the online addendum. A graphical summary of an example system which includes multiple versions of JUnit is provided in Figure 4.

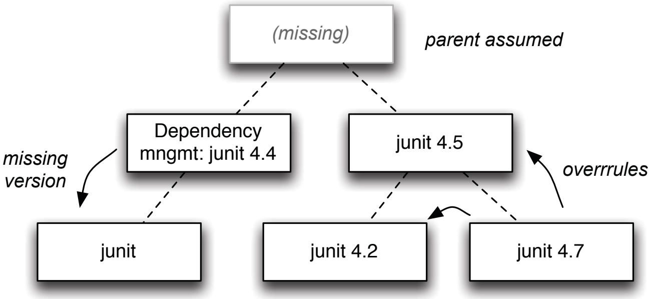

Figure 4. An example of a system with multiple versions of the same library. A rectangle represents a single pom file.

Since the dependencies section in a Maven dependencies file can contain both publicly available and internal dependencies, internal dependencies were removed by matching dependency names with project names. After this, results were manually checked to remove false positives. This resulted in a list of third-party library dependencies per snapshot for each system. Next, binary, source and JavaDoc jars of all collected dependencies were downloaded from the central Maven repository, if available. After downloading, all jar files were automatically extracted. For descriptive statistics of collected dependencies, see Table II.

For each system snapshot, source code of the system and source code of all used third-party libraries were combined in a single folder. The Software Analysis Toolkit (SAT) of the SIG was used to calculate a wide range of metrics related to size in lines of code (LOC), complexity by means of the McCabe value as well as call graph information. The result of this process is a data file containing metrics on method-level for each snapshot of systems and third-party library code that is called from this system. Package, class and method names are stored separately to make aggregation of metrics to these levels possible. By comparing versions and names of third-party library dependencies between snapshots it is possible to detect additions, removals, and version changes of these libraries.

In the next section we consider different variables to include in a library stability metric.

VII. Metric Ingredients

There are several possible metrics and measurement methods that could be considered for inclusion in a metric of the stability of library implementation and interface. The criterion we maintain for a library stability metric is that it should be representative for the amount of work that is required when library developers update a certain library to a newer version. Included metrics should therefore have a rationale that is consistent with this criterion. Since we want to investigate the stability of both implementation and interface of a library we also consider metrics that are an indicator for the amount of implementation "churn". We expect that even though these changes do not become visible at the public interface, there is still a potential amount of rework effort required due to a potential change in behavior.

Table III summarizes change characteristics for Commons Logging and Commons Collections. Commons Collections has more different snapshots and has more methods that are removed and deleted in each snapshot than Commons Logging and Commons Logging is therefore considered to be more stable.

Library	*S*	*Unique methods (diff)*	*Total McC. (diff)*	*Total LOC (diff)*
Logging	1	263	454	1521
	2	301 (+75, -37)	653 (+199)	2818 (+1297)
	3	311 (+11, -1)	667 (+14)	2918 (+100)
Collections	1	1504	2580	9969
	2	1398 (+0, -106)	2395 (-185)	9482 (-487)
	3	3357 (+2218, -259)	5583 (+3188)	18199 (+8717)
	4	3821 (+568, -104)	6503 (+920)	20452 (+2253)
	5	3945 (+125, -1)	6749 (+246)	21207 (+755)
	6	3945 (+0, -0)	6749 (+0)	21207 (+0)

Table III
Example of the differences in number of methods, total McCabe and total LOC for two Apache Commons libraries

A. *Candidate Variables*

The candidate variables to include are the following:

1) Unit removals: Units (methods, classes or packages in Java) that are being removed from a public interface require rework from the developers that call these units. Therefore, counting the number of removed units per snapshot is a good indicator for the stability of an API. We detect renamed/moved units as units that are removed first and added later. This is acceptable since a unit's rename or move also requires rework effort and can therefore be counted as a unit removal.

2) Unit additions: Units that are added to a library do not influence the stability of the existing interface directly but could serve as an indirect indicator of the amount of effort that is spent on extending the library. Examples of method additions and removals in two libraries can be found in Table III.

3) LOC an McCabe changes: Changes in McCabe and LOC values serve as indirect measure for the amount of work going on in a library since implementation can change while an interface stays constant. It is also not possible to make a distinction between changes that alter external behavior and changes which do not alter behavior of the library (such as refactorings) by looking at the McCabe value or the LOC alone. We nevertheless believe that these metrics provide an indication of the amount of work performed in a certain library.

4) Parameter changes: In principle, a change in the signature of an existing API method always requires rework.

Therefore, a separate analysis of signature changes could be included in our metric. However, signature changes will usually go hand in hand with method body changes. We will therefore ignore signature changes for the moment and focus on size-related measurements instead.

We incorporate these variables in our metrics as can be seen in Section VIII. First, we discuss how we weight historical values of the same metric and how we incorporate usage information in our metrics.

B. Weighting Measurements Historically

Because multiple historic values of the same metric exist, a method is needed to aggregate multiple measurements in time to a single value. After calculation of the difference in each metric between two snapshots we obtain a set of absolute differences for each unit in a system. To reduce this set to a single number, we use a historic weighting scheme to put emphasis on more recent snapshots. For our analysis, we choose a geometric series to weight each snapshot: a metric in snapshot s (counting backwards) is weighted with $1/2^{s-1}$. This reflects our belief that changes made more recent in time weight more heavily than changes made longer ago. This is illustrated in Figure 5.

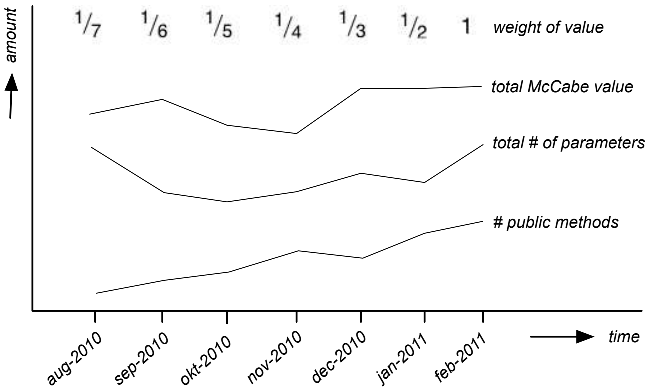

Figure 5. An example of the evolution of the number of parameters, the McCabe value and the LOC for a unit. Above the X-axis is a weighting scheme that puts more emphasis on recent values.

C. Weighting Measurements via API Usage

Besides weighting each snapshot differently we also give weight to each unit in a snapshot, for which we use the frequency of use as a weight. This means that more frequently used units have more influence in the final metric. This way, changes in units that are called frequently are emphasized and have a greater influence on the final metric of a library than changes in methods that are never called. Section IX-A shows an overview of these frequencies of use.

We assume that methods are part of the public interface of a library if they can be called from a system using this library, regardless of the mechanism used to ensure this. This means that we ignore the mechanism provided in Java to make a distinction between public, package-private, private and protected methods and classes and we use the actual usage of methods inside a corpus of industrial systems as a weight. Units that are called frequently will have more influence in the final metric than units which cannot be called externally, which receive a weight of 0.

In the next section, we present our metrics which incorporate previously discussed variables, usage information of libraries and our historical weighting scheme.

VIII. Metric Definitions

We propose 4 metrics, which are displayed in equations 7 to 10 and are defined as follows:

WRM: The weighted number of removed methods
CEM: The amount of change in existing methods
RCNO: The ratio of change in new to old methods
PNM: The percentage of new methods

The metrics can be explained as follows. The WRM uses the weighted number of removed methods from an interface as a value for R_x in equation 6, which is a measure for interface stability since the removal of units from a public interface becomes immediately visible for users of a library. The more times a removed method is being used the more it increases the WRM.

CEM gives an indication of the amount of change in existing methods. It can give an impression of the activity and “volatility” of the development of a library.

RCNO uses the ratio in metric difference between new and existing units which can be used to determine the amount of work being performed in new units relative to old ones. This ratio is smaller than 1 if more work is being performed in old methods and is greater than 1 if more work has been done in new methods. It is useful for determining whether a system is in a state of maintenance or active development. Developers can only spend a limited amount of time per release on new features and maintenance and the time spent on one activity cannot be spent on the other.

PNM calculates the percentage of new methods that have been added in each snapshot and can provide information on the expansion rate of an API.

Equations 1 to 3 are functions which select appropriate units: U_o (old) is the collection of units that are in snapshot s and in snapshot $s+1$. Similarly, U_n (new) is the collection of units which are new in snapshot $s+1$ compared to snapshot s. U_r (removed) are the units that are in s but not in $s+1$. Equation 5 states that the combined delta between two snapshots of all units is the difference between a metric value in snapshot $s+1$ and s, weighted by the times each unit is being called. The result is normalized by total weight of units in $\Delta U(s, s+1)$. ΔU_n is the difference between 0 and the metric value for each new unit while ΔU_o is the difference between the metric value in s and $s+1$. Equation 5 expresses the absolute weighted difference of each unit in snapshot $s+1$ compared to s per snapshot.

$$U_{o(s,s+1)} = U_s \cap U_{s+1} \tag{1}$$

$$U_{n(s,s+1)} = U_{s+1} \setminus U_s \tag{2}$$

$$U_{r(s,s+1)} = U_s \setminus U_{s+1} \tag{3}$$

$$hw(s) = \frac{1}{2^{s-1}} \tag{4}$$

$$\Delta U(s, s+1) = \frac{1}{\sum\limits_{u \in \Delta U} w_u} \sum_{u \in \Delta U} w_u |M_{u,s+1} - M_{u,s}| \tag{5}$$

$$R = \sum_{s=1}^{|S|-1} hw_s R_x \tag{6}$$

$$\mathit{WRM} = \sum_{u \in U} w_u U_r \qquad \mathit{CEM} = \Delta U_o \tag{7, 8}$$

$$\mathit{RCNO} = \frac{\Delta U_n}{\Delta U_o} \qquad \mathit{PNM} = \frac{|U_n|}{|U_o| + |U_n|} \tag{9, 10}$$

Symbol	*Explanation*
u	A specific unit
U	The set of all units in a system
s	A specific snapshot number (snapshots are ordered on date, are numbered backwards and the latest snapshot gets number 1)
S	The set of all snapshots of a system
U_s	All units in snapshot s
U_o (old)	All units in $s+1$ also in s
U_n (new)	All units in $s+1$ but not in s
U_r (removed)	All units in s but not in $s+1$
$hw(s)$	The historical weight of a snapshot
$M_{u,s}$	The value of metric M of unit u at snapshot s
w_u	The weight of a unit (the total times the unit is used in our dataset)
R	Generic placeholder metric function
R_x	A specific metric (WRM, CEM, RCNO or PNM) to be used in the placeholder function

Table IV
EXPLANATION OF USED SYMBOLS IN EQUATIONS 1 TO 10

Eventually, all snapshots are combined in a single metric formula by summing over the separate metrics and giving less weight to snapshots further away in time. Ultimately, these metrics make it possible to aggregate values at the unit level to a single value at the system level while capturing changes through time and weighting for frequency of use and recency of snapshots. For an explanation of symbols used in equations 1 to 10, see Table IV.

IX. APACHE COMMONS FINDINGS

With all metric definitions in place, we can analyze the metric values for Apache Commons in the context of our suite of 140 benchmark systems. We start by analyzing which library methods are used most frequently, followed by a discussion of the actual metric values.

A. Apache Commons API Usage

In Table V, the most frequently called methods per library are shown. As can be seen in this table, the most frequently used method constitutes a large percentage of the total number of calls to a library. Also, only a small percentage of methods is actually called from our dataset. Most methods are used only once or not at all. This is also illustrated in Figure 6, which gives a visual impression of the spread and concentration of method calls through each library. We use this information to determine the weight for each unit (w_u) in equation 5 in the previous section.

Library	*Method name*	*# Calls*	*%*
Commons Logging	Log.info	15114	28.54%
Commons Lang	StringUtils.isNotEmpty	10033	10.75%
Commons Collections	NotPredicate constructor	2461	30.14%
Commons Beanutils	DynaBean.get	875	30.71%
Commons IO	FileUtils.readFileToString	835	11.55%
Commons Httpclient	HttpClient.executeMethod	428	9.23%
Commons Codec	Base64.decodeBase64	124	30.77%

Table V
THE NUMBER OF TIMES THE MOST FREQUENTLY USED METHOD IS CALLED AND THE PERCENTAGE OF THE TOTAL NUMBER OF CALLS

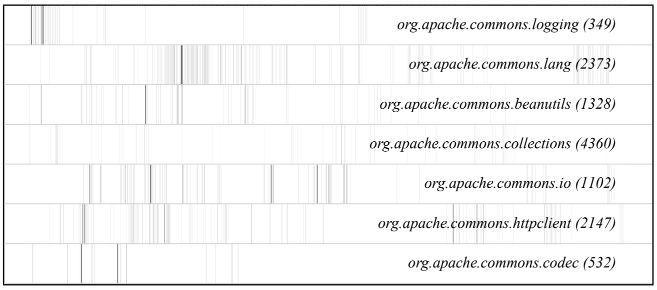

Figure 6. The relative distribution of calls to methods of third-party libraries. A darker shade of grey means a certain method is called more. Methods are sorted on method and package name. This figure does not show method names or the exact number of times each method is called but gives a visual impression of the spread and relative usage of library methods. In parentheses is the total number of methods in that library.

B. Metrics

The results for the four metrics are shown in Table VI. An absolute value and a rank is provided for each metric. The metric values are dimensionless and range from 0 to infinity; smaller values indicate greater stability.

Table VI shows that there are 4 systems with a value of 0 for WRM. This indicates that there have been no methods in those libraries which were used in our dataset and were removed from a next snapshot of the library. The WRM can be seen as an indicator for the absolute number of methods which will have to be adapted due to method removals from these libraries.

Since multiple systems have the same score for WRM, the value of CEM can provide additional information. Of all systems with a 0 for WRM, Commons Logging is the system with the highest score for CEM (0.0124). This means that the amount of change in existing methods is relatively

Library	*WRM (Rank)*	*CEM (Rank)*	*RCNO (Rank)*	*PNM (Rank)*
Commons Logging	0 (1)	0.0124 (2)	0 (1)	0.2669 (2)
Commons Beanutils	267.25 (3)	1.257 (7)	10.175 (3)	0.4931 (6)
Commons Codec	31 (2)	0.7 (6)	1.8833 (2)	0.5083 (7)
Commons Httpclient	0 (1)	0.1239 (4)	146.84 (4)	0.1937 (1)
Commons IO	0 (1)	0.0484 (3)	273.16 (5)	0.4281 (5)
Commons Lang	0 (1)	0.3456 (5)	481.09 (7)	0.3256 (3)
Commons Collections	1062.0 (4)	0.0077 (1)	339.65 (6)	0.3715 (4)

Table VI
VALUES AND RANKS FOR THE FOUR METRICS

low in Commons Logging compared to other systems. Commons Logging also scores a 0 for RCNO, which can mean three things: (1) there is no difference in existing methods, (2) there is no difference in new methods (only empty method bodies have been created) or (3) all changes occur in methods which are never called. From this metric, it is not possible to distinguish between these cases, but further inspection showed that all metric differences occured in methods which were never called. Commons Logging scores 0.2669 for PNM, which is the second place. This indicates that a relatively small number of new methods are added in each snapshot. Metrics should not be interpreted as percentages or amounts directly since snapshots have been weighted differently with the use of a historical weighting scheme.

X. SCENARIOS

To further interpret the results of our metrics we present three scenarios and an example of the application of our metrics.

Consider the following scenarios, in which a software developer or project manager needs to:

1) decide whether or not to depend on a certain third-party library to perform a certain function;
2) decide whether or not to create a wrapper around a third-party library to encapsulate dependencies on this library to reduce risks;
3) determine if a library is in a state of maintenance or active development.

We demonstrate how to apply our metrics and how these metrics can help in making a decision in these scenarios.

Assume that a developer wants to know if he should include the Commons Codec library in their software project or if he should write his own codec methods. There are several trade-offs to consider when making this choice. Commons Codec is a library which contains highly specialized functionality and it is likely to take a large effort to rebuild. It also requires a large amount of specialized knowledge which the developer may not have.

On the other hand, if the implementation or interface of the library is unstable, including this library may become a risk to the stability of the software itself. Other code may need to be rewritten to adapt to the new library interface. When developing the functionality internally, the developer also has greater control over added features and maintenance of the code. Table VI shows that Commons Codec scores 31 for metric WRM, which is the second place. This means that the API of Commons Codec tends to stay relatively stable regarding the removal of frequently used methods.

Another consideration to make is whether the latest version of a library already contains required functionality or if it is not included or completed yet. PNM shows the percentage of new methods in each snapshot. If required functionality is already included then it is not necessary to wait for the latest updates but when a new piece of functionality is required which is not implemented yet, such as a new video codec, it may be worth the wait. In the case of Commons Codec, the library has the highest score (0.5083), which indicates a large degree of new methods in each snapshot. A similar reasoning can be followed when looking at metric RCNO, which shows the ratio of work in new to old methods. Commons Codec has a score of 1.8, which shows that there is more work going on in new methods than in old ones. To obtain a more detailed picture of the evolution of a system, scores and metrics per snapshot can be obtained, which are shown in Table VII.

s	$\lvert U\rvert$	$\lvert U_n\rvert$	ΔU_n	$\lvert U_o\rvert$	ΔU_o	$\lvert U_r\rvert$	hw_s	*metric values*
4	215	-	-	-	-	-	-	-
3	318	107	1.32	211	0.7	4	$1/4$	0, 0.7, 1.88, 0.33
2	373	80	0	293	0	25	$1/2$	62, 0, 0, 0.21
1	503	130	0	373	0	0	1	0, 0, 0, 0.25

Table VII
DETAILS OF THE METRICS FOR APACHE COMMONS CODEC

Table VII shows that there were 4 methods removed in the first snapshot, there were 25 methods removed in snapshot 2 and there were 0 methods removed in snapshot 1 (latest). The percentage of new methods is relatively high in snapshot 3: 33%. In the latest two snapshots, ΔU_n is 0. The release notes of Commons Codec 1.5 (snapshot 2) state that new methods were added to the public interface, but apparently these are not yet used in our dataset. Similarly, ΔU_o is 0 for the latest two snapshots, which means that there has not been change in methods that are being called in our dataset.

To get even more information, names and metric values for methods in U_n, U_o and U_r for each pair of subsequent snapshots can be obtained. With this information, identification of most frequently used and most changing methods becomes possible. We do not show this list here but this information could be included when using our approach in practice. This way, identification of potential issues in removed or changed methods methods in third-party libraries becomes possible.

Our metrics can be used to get a clearer picture of the historic stability of a library. Although we do not expect that a small difference in these metrics has a large effect on a system which uses this library, an instable library can pose a problem in the long run, when deferred updates are accumulated as “technical debt”. Our metrics also provide a way to test whether the reputation of a library in the open source community is really deserved, which is for a large part based on provided functionality but may also be influenced by the stability of a library in the past and the way the library deals with breaking changes. We try to catch these aspects in our metrics. Our metrics can provide more practical help when a decision needs to be made to create a wrapper around a library which encapsulates changes in that library.

XI. Discussion

This paper introduced several new concepts and metrics to measure the stability of the implementation and interface of a library through time. In this section, we discuss implications and limitations of our study and we discuss directions for future work.

A. Used metrics

In this paper we use McCabe as metric value in our formulas. Instead, different metrics and units can be used. For example, it could be useful to calculate the number of methods per class as metric and the usage frequency of each class as weight. This would provide information on a higher level of abstraction than McCabe or LOC values per method and would give an idea about the evolution of the size of each class through time.

The size of libraries differs significantly and we do not know how big the effect of system size is. In future work, we plan to investigate this effect of size and to adjust for this effect. We expect that a large part of stability is independent of system size and our metrics can therefore also be used without this normalization.

It can be difficult to interpet dimensionless and rangeless numbers directly, which can only be properly done after comparison or benchmarking. For this reason, these metrics should be be calculated for a greater set of libraries. In future work, we plan to calculate our metrics on the complete Maven repository, containing over 300,000 artifacts with multiple versions. Such a large collection of metrics enables a more detailed analysis of distributions and percentiles.

B. Using historical data

We use historical data from software repositories to calculate a metric which gives an impression of the "volatility" of the interface and implementation of a library. To achieve this, only historical data is used and a question is to what extent this can be used to predict future developments. With this approach, unlikely and unforeseen trend breaks will not be detected but it gives a good indication of the historical trend of a library. Assuming that the same development team keeps working on the code, developer behavior is expected to follow a similar trend as in the past.

Another issue with our approach is that code of historical snapshots is required to calculate a metric. This limits the applicability of our method to open-source libraries or libraries of which source code is otherwise available. By performing our method on the most frequently used third-party libraries we hope to provide practical knowledge and pointers to best practices of API stability which can be applied to other libraries as well.

Ideally, more fine-grained data should be used for analysis such as change sets per commit from a version control system. This information is not always available, however, and metrics depending on this information would limit their applicability to systems with access to their version control systems. Our metrics do not have this requirement.

C. Estimating rework effort

To properly estimate the amount of rework effort required after the upgrade of a library, an experiment needs to be performed in which the impact of changes in our dataset is investigated. Differences between metric values of units with connections to third-party units and units without connections could be compared. This way, an estimate in terms of time or money could be roughly calculated. This experiment would also provide a validation of our metrics since we defined the effect of library instability to be a large amount of required rework that has to be performed after upgrading that library. If libraries with high scores for our metrics also cause larger differences in metric values in units with links to third-party units than in units without these links, then our metrics have been validated against a real-world dataset, assuming that the chosen metric difference is a good indicator for the amount of expected rework.

D. Transitive calls and dependencies

In this paper, we ignored transitive third-party references and transitive third-party method calls. When method m_1 calls method m_2 which in turn calls a third-party method m_3, then in our current analysis, only m_2 would be impacted by a potential change in m_3. In future work, transitive method calls could be included using a similar weighting scheme as applied to snapshots: methods that are closer to a third-party method in the call chain are potentially more influenced by a change than methods further away.

XII. Threats to validity

A. Internal validity

Our simplified Maven dependency resolution system always chooses the latest version of a library in case of a conflict. The real Maven dependency resolution system is somewhat more complex and contains advanced heuristics as well. It is therefore possible that we included the wrong library version in certain snapshots, but we expect that the occurence of this issue is negligible in our dataset.

Our dependency framework also potentially misses changes in the number of parameters of methods since this can be confused with an added overloaded method in the next snapshot. We did not investigate the number of parameters as metric value in this paper but future work using this metric value should investigate this issue further.

B. Construct validity

We chose to use a geometric series as weight for previous snapshots since this series has the property that the ratio of subsequent terms is constant and that the sum is finite. An alternative weighting scheme would change the final outcome of the metric. We believe that our choice is justifiable, since snapshots further away in time are deemed less important than more recent snapshots. More research is needed to understand alternative metric schemes and their impact on metric outcomes.

The choice of metric is also of great importance in our metric. In this paper, we chose to use the difference in McCabe value. This has the consequence that when the LOC of a unit changes without adding a decision point (e.g. `if`, `while`, `case`), no difference is detected. To investigate the differences, the same metrics should therefore also be calculated with the LOC as metric. Our equations provide a general framework in which different metrics and units can be used. More research has to be performed before these alternatives can be used in practice.

C. External validity

Our dataset used to obtain frequencies of use is considered to be representative for a wide range of industrial areas, such as insurance, banking, logistics and government. However, analyzed systems were all written in Java and use Maven as build configuration tool. We estimate that reuse of existing open-source components is more prevalent in the Java community than in communities of other programming languages and therefore a possible bias may exist which overestimates the frequency of use for third-party library methods. We do not believe that restricting our dataset to systems which use Maven as build tool introduces a bias since it only helps to identify exact dependencies and version numbers. Other build tools, such as Apache Ant, do not explicitly state version numbers but still have the possibility to include third-party libraries.

XIII. Conclusion

In this paper, we presented four stability metrics which calculate the stability of the public interface and implementation of a library based on the weighted number of removed methods, the change in metric values in existing units, the ratio between change in new and old methods and the percentage of new methods per snapshot. We investigated results of our metrics and showed an example of their application to an open source library.

The contributions of this paper are the following:

- A case study of the upgrade of third-party libraries in a commercial software system which shows several issues associated with the use of these libraries;
- A framework for fact extraction and analysis of library dependencies in Java projects built with Maven;
- A proposal for four library stability metrics, incorporating weighting schemes for recency and for API usage;
- Instantation of the metric framework to the Apache Commons libraries

Furthermore, we have identified a number of scenarios explaining how these metrics can be used. In our future work, we aim at benchmarking the proposed metrics, through an analysis of the full Maven Central repository.

References

[1] B. Dagenais and M. P. Robillard. SemDiff: Analysis and recommendation support for API evolution. In *Proc. of the 31st Int. Conf. on Software Engineering*, ICSE '09, pages 599–602, 2009.

[2] D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proc. of the 21st IEEE Int. Conf. on Software Maintenance*, ICSM '05, pages 389–398, 2005.

[3] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proc. of the 2010 ACM-IEEE Int. Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, 2010.

[4] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proc. of the 27th Int. Conf. on Software engineering*, ICSE '05, pages 274–283, 2005.

[5] 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, 2011.

[6] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining trends of library usage. In *Proc. of the joint Int. and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 57–62, 2009.

[7] J. H. Perkins. Automatically generating refactorings to support API evolution. In *Proc. of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 111–114, 2005.

[8] S. Raemaekers, A. van Deursen, and J. Visser. An analysis of dependence on third-party libraries in open source and proprietary systems. In *Sixth Int. Workshop on Software Quality and Maintainability*, SQM '12, 2012.

[9] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. of the 2008 23rd IEEE/ACM Int. Conf. on Automated Software Engineering*, ASE '08, pages 327–336, 2008.

[10] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal API usage patterns. In *26th IEEE/ACM Int. Conf. on Automated Software Engineering*, ASE '11, pages 456–459, 2011.

[11] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. of the 2006 Int. workshop on Mining software repositories*, MSR '06, pages 54–57, 2006.

[12] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In S. Drossopoulou, editor, *ECOOP 2009 Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343, 2009.