A Structured Approach to Assess Third-Party Library Usage

Veronika Bauer Lars Heinemann
Technische Universität München, Germany
{*bauerv, heineman*}*@in.tum.de*

Florian Deissenboeck
CQSE GmbH, Germany
deissenboeck@cqse.eu

***Abstract*—Modern software systems build on a significant number of external libraries to deliver feature-rich and high-quality software in a cost-efficient and timely manner. As a consequence, these systems contain a considerable amount of third-party code. External libraries thus have a significant impact on maintenance activities in the project. However, most approaches that assess the maintainability of software systems largely neglect this important factor. Hence, risks may remain unidentified, threatening the ability to effectively evolve the system in the future. We propose a structured approach to assess the third-party library usage in software projects and identify potential problems. Industrial experience strongly influences our approach, which we designed in a lightweight way to enable easy adoption in practice. We present an industrial case study showing the applicability of the approach to a real-world software system.**

***Keywords*-software reuse, API, library, software maintenance**

I. Introduction

Reuse with software libraries plays a central role in modern software development—instead of writing a complete software system from scratch, significant parts of its building blocks are reused from third-party libraries. Especially for widely-used platforms like Java, a considerable amount of reusable libraries with a large variety of functionality is available in code repositories on the Internet. In a recent study on reuse in Java open source projects we found that for almost half of the projects the amount of reused code exceeded the amount of newly developed code [1].

However, library reuse comes at a cost: Included libraries can significantly impact the maintainability of a software system. Often, projects use a number of different libraries and the code is highly entangled with their APIs. This poses multiple risks to the evolution of the software. First, libraries continuously evolve. New releases provide added functionality and bug fixes. In many cases, it is desirable to migrate a software system to the latest stable release of a library, especially in case of critical bugs such as security flaws. However, migration can cause considerable maintenance effort, as backward-compatibility may not always be ensured. Second, a library might be still unstable and introduce bugs into the software, which could be difficult to find and hard to fix. Third, the provider's support or maintenance of a library might be discontinued, such that the library consumer can no longer expect fixes for critical bugs. Finally, the license of a library or the legal constraints in a project may change, forcing the project to stop employing the library. Again, a potentially costly replacement with a different library or an own implementation of the reused functionality is required. Unfortunately, most existing approaches that aim at assessing the maintainability of a software system primarily focus on the project's own code. The included libraries are often disregarded, missing important aspects affecting maintainability.

Problem: A plethora of external software libraries form a significant part of modern software systems. Consequently, these systems contain a considerable fraction of code developed and maintained by third parties. Therefore, external libraries and their usage have a significant impact on the maintenance of the including software. Unfortunately, third-party libraries are often neglected in quality assessments of software, leading to unidentified risks for the future evolution of the software.

Contribution: Based on industry needs, we propose a structured approach for the systematic assessment of third-party library usage in software projects. It can be applied to support specific maintenance decisions as well as to monitor the project's state of reuse over time. The approach is supported by a comprehensive assessment model relating key characteristics of software library usage to development activities. The model defines how different aspects of library usage influence the activities and, thus, allows to assess if and to what extent the usage of third-party libraries impacts the development activities of a given project. Furthermore, we provide guidance for executing the assessment in practice, including tool support for a pre-selection of important libraries and multiple automated static code analyses. We evaluate the approach with a case study involving an industrial software system of 3.5 MLOC including about 90 external libraries.

Outline: The remainder of the paper is organized as follows: Section II introduces the industry partners of this work. Section III presents the assessment model, whilst Section IV details the assessment process. Section V describes the tool support provided for the analysis. Section VI presents design and results of the case study. The following Sections VII and VIII discuss results and threats to validity. Section IX gives an overview on related work before Section X concludes the paper and identifies future work.

II. Industry Partners

This work involves two industry partners: The model was designed in cooperation with CQSE GmbH[1]. azh Abrechnungs- und IT-Dienstleistungszentrum für Heilberufe GmbH[2] was our partner for the case study. This section briefly introduces the two companies.

The CQSE GmbH was founded in early 2009 as a spin-off of the competence center for Software Quality and Maintenance at the chair for Software & Systems Engineering of the Technische Universität München. It provides consulting services for software quality assurance. In particular, it helps customers in applying novel techniques like clone detection and architecture conformance analysis to ensure long-term maintainability of their software systems.

The azh Abrechnungs- und IT-Dienstleistungszentrum für Heilberufe GmbH is one of the largest provider for billing and IT-services for professional health care providers in Germany. With 550 employees they provide support for 20,000 customers.

III. Assessment model

The proposed assessment model is inspired by activity-based quality models [2] that offer a soundly structured and precise way of expressing quality factors and their mutual dependencies. In this model, we follow the terminology coined by Kitchenham et al. where *entities* "are the objects we observe in the real world" and *attributes* are "the properties that an entity possesses" [3]. We extend this by allowing an attribute to be attached to multiple entities and adapted the meta-model from [2], as shown in Figure 1. This is necessary as several relevant library attributes emerge only from the relation between entities. Figure 2, which shows the concrete instantiation of the meta-model, contains examples: for instance, the attribute ENTANGLEDNESS is attached to the entities System and Library as it models a characteristic that involves both.

Entities are structured in a hierarchical manner to foster completeness. The combination of one or more entities and an attribute is called a *fact*. Facts are expressed as [Entities | ATTRIBUTE]. A fact has an *assessment type*, which can be a manual assessment by an expert, an automatic analysis, or semi-automatic as a combination of the above. To express the impact of a fact, the model relates the fact to a development *activity*. This relation can either be positive, *i. e.*, the fact eases the affected activity, or negative, *i. e.*, the fact impedes the activity. In Figure 2, for example, the ENTANGLEDNESS of System and Library has a negative impact on all maintenance activities whereas the protection against security-relevant attacks as well as the legal aspects of the distribution of the system are not affected by this fact. *Impacts* are expressed as [Entity | ATTRIBUTE] $\xrightarrow{+/-}$ [Activity]. Each impact is backed by a *justification*, which provides the rationale for its inclusion in the model.

So far, the model constitutes a *Definition Model* in the sense of [4] as it defines quality aspects and their relations. To become an assessment model that can be used to assess a specific situation, it needs to be enriched with metrics and a measurement method. We provide this with the three-value ordinal scale {*low, medium, high*} that is used to *quantify* facts. The assessment of the facts is mainly a manual activity performed by an expert who bases the judgement on a set of metrics that may be determined automatically. Table I shows the metrics used in our model.

To *assess* the impact on the activities, we use the three-value scale {*bad, satisfactory, good*}. If the relation between a fact is positive, there is a straight-forward mapping from *low* → *bad, medium* → *satisfactory, high* → *good*. If the fact [Library | PREVALENCE], for example, is rated high, the effect on the activity *migrate* is good as the impact relation is positive [Library | PREVALENCE] $\xrightarrow{+}$ [Migrate] (see Figure 2) as a high prevalence of a library usually gives rise to alternative implementations of the required functionality. If the impact relation is negative, the mapping is turned around: *low* → *good, medium* → *satisfactory, high* → *bad*. A high [Library,System | ENTANGLEDNESS], for example, results in a bad effect on the activity *understand* as the relation is negative: [System, Library | ENTANGLEDNESS] $\xrightarrow{-}$ [Understand]. A high entangledness tends to cause difficulties to clearly understand how a library is to be used. Especially scatteredness of method calls could hamper understanding.

The assessment of a single library thus results in a mapping between the activities and the {*bad, satisfactory, good*} scale. To aggregate the results, we simply count the occurrences of each value at the leaf activities. Hence, the assessment of a library finally results in mapping from {*bad, satisfactory, good*} → $\mathbb{N}_0$. We deliberately do not use an aggregation that results in a single number to avoid comparing apples with oranges.

A. Activities

With one exception, the activities included in the model are typical for maintenance. The following paragraphs briefly describe each of them.

Modify: Modifying a system means to change it to *e. g.*, fix a bug or add new functionality.

Understand: Understanding a system is critical for all maintenance tasks. Developers need to understand the structure and functionality of a system before they can start to extend or change it.

Migrate: Migration is the process of exchanging a library with a newer version or a different library. When migrating a software system to another library, it is usually important that the new library comprise the same functionality as the old one.

[1] http://www.cqse.eu/

[2] http://www.azh.de/

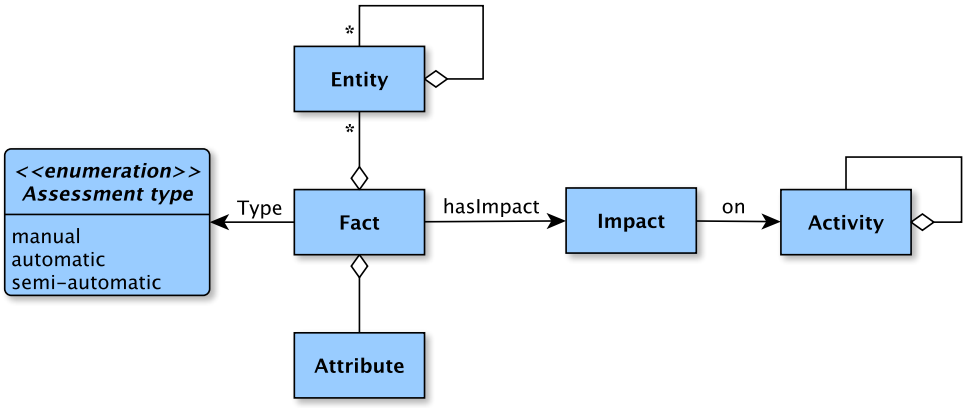


Figure 1. The meta-model of the assessment model.

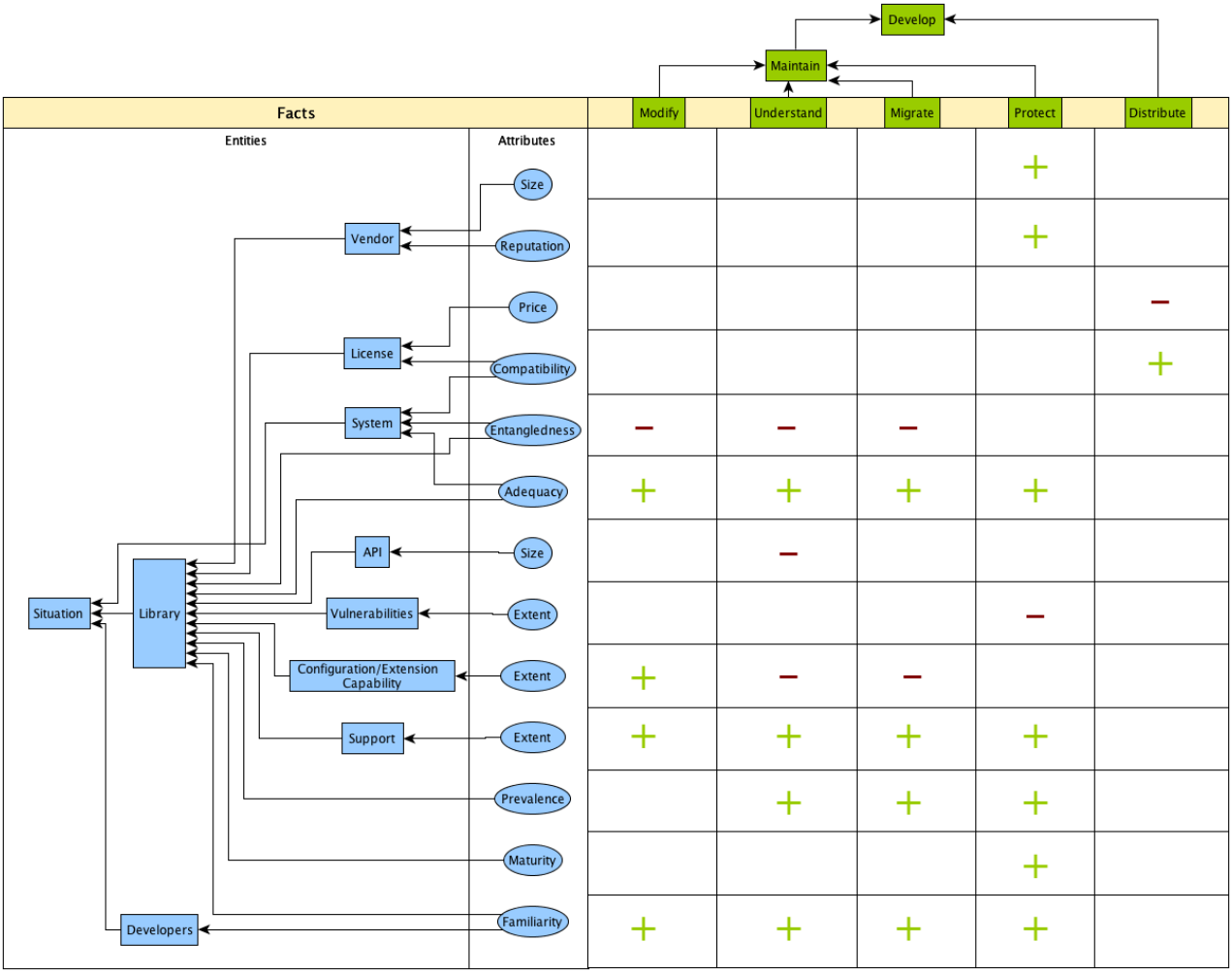


Figure 2. Instantiated assessment model with facts, development activities, and impacts between them.

Protect: Often, security issues surface during the productive usage of a software system. It is important to fix these during maintenance, *e. g.*, by migrating the system to an updated library version.

Distribute: Distributing a software system refers to any kind of proliferation, such as providing the system open source on the Internet or shipping the product to a commercial customer. This activity is not directly linked to maintenance. However, third-party libraries impact it.

B. Metrics

The model quantifies each fact with one or more metrics. The list of metrics, their description and assignment to facts are shown in Table I. As an example, to quantify the extent of vulnerabilities of a library, we measure the number of known critical issues in the bug database of the library. Some of the facts cannot be measured directly, as they depend on many aspects. For instance, the maturity of a library cannot be captured with a single metric but must be judged by an expert. We do not employ an automatic, *e. g.*, threshold-based, mapping from metric values to the {low, medium, high} scale but fully rely on the experts capabilities.

C. Impacts

Impacts define how facts influence activities. A justification for each impact provides a rationale for the impact which increases confirmability of the model and the assessments based on the model. Note that a fact might have a positive impact on one activity whilst negatively impacting another one. Due to space constraints, we give one example of an impact per activity. The complete list can be obtained from our website[3].

[Extension/Configuration Capability | EXTENT] $\xrightarrow{+}$ [Modify] A high extension capability of an external library positively impacts modifications, such as adding new features, because it increases the chances that they can be accomplished with the same library.

[Extension/Configuration Capability | EXTENT] $\xrightarrow{-}$ [Understand] Whilst high extension capability positively impacts modifications, it hinders understanding. The reason is the high complexity caused by the flexibility of extension mechanisms which make it harder to understand how to use the library.

[Extension/Configuration Capability | EXTENT] $\xrightarrow{-}$ [Migrate] A high capability for extension and configuration inhibits migration as it is less likely that alternatives provide the same flexibility.

[Vendor | REPUTATION] $\xrightarrow{+}$ [Protect] The reputation of the library vendor positively influences protection of a system, as a renowned vendor can be expected to provide critical updates in a timely manner.

[License,System | COMPATIBILITY] $\xrightarrow{+}$ [Distribute] Characteristics of third-party libraries also impact the distribution of a system, *e. g.*, low compatibility of licenses can block the distribution of a system.

[3]http://www4.in.tum.de/~ccsm/library-usage-assessment/

IV. Assessment Process

Our assessment process provides guidance to operationalize the model for assessing library usage in a specific software project. When assessing a real-world project, the sheer number of libraries require a possibility to address the *most relevant* libraries first. Therefore, the first step of the process structures and ranks the libraries according to their entangledness with the system. This pre-selection directs the effort of the second step of our process: the expert assessment of the libraries. The last step describes how to generate an assessment report from the aggregated results.

A. Ranking and Pre-selection

In the first step of our assessment process, a set of automatic static analyses are run on the project to be assessed. Their goal is to objectively determine the degree of entangledness between external libraries and the system. This step ensures the applicability of our approach in practice, as extracting these values by hand is unrealistic for real world projects. Furthermore, the extracted metrics help to identify the most significant candidates for the expert assessment and therefore decrease the effort of the system review.

1) Ranking: To rank the libraries according to their entangledness with the system, we extend work presented in [5] and determine the following values for all libraries: The number of *total method calls* to a library allows to rank all external libraries according to the strength of their direct relations the system. The value is computed for the entire system hierarchy to allow a drill down from system level to class level. This way, the point of impact can be explored precisely. This is relevant for cases, in which participants of the assessment wish to understand in detail where a library affects their system. The number of *distinct method calls* to a library adds information about the implicit entangledness of libraries and system. It allows to understand how difficult a migration could be. The granularity of the computed value is the same as for the total number of method calls. The *scatteredness of method calls* to a library describes whether the usage of the library is concentrated to a specific part of the system, or scattered across it. The value is computed based on the package structure of the system (for further details see Section V). The *percentage of affected classes* gives a complementary overview about the impact a migration could have on the system. This value is independent from the system structure.

The results of these analyses are presented as an HTML document, which provides the possibility to inspect the raw data, as well as detailed insights via the drill-down. The data for each metric is presented in a tabular way with the libraries ordered descendingly according their scores.

Table I
FACTS AND ASSOCIATED METRICS OF ASSESSMENT MODEL

Fact	Metric	Description
[Support | EXTENT]	Expert assessment	Availability of support and training
[Vendor | SIZE]	Headcount	The number of contributors is determined *e. g.*, from the vendor's website
	Sales Volume	The sales volume is determined *e. g.*, from the vendor's website
[Vendor | REPUTATION]	Expert assessment	Reputation as perceived by an analyst/domain expert
[License | PRICE]	Price	Price for a redistribution license
[Library | PREVALENCE]	#Books	Number of search results on Amazon
	#Google Hits	Number of search results on Google
	#Job Advertisements	Number of search results on jobpilot.de
[Library | MATURITY]	Expert assessment	Development status (*e. g.*, development, inactive, stable)
[API | SIZE]	#API types	Number of types listed in the Javadoc
[Vulnerabilities | EXTENT]	#Known critical issues	Number of security issues in the bug database
[Extens./conf. capabilities | EXTENT]	Expert assessment	Availability and characteristics of extension and configuration features
[Library,System | ENTANGLEDNESS]	#API calls	Number of API method calls
	#Distinct API methods	Number of distinct API methods called
	%Affected classes	Fraction of classes in the system with API method calls
	Scatteredness of API calls	Degree of scatteredness of API calls regarding the package structure
[Library,System | ADEQUACY]	Expert assessment	How well does the actual use correspond to the intended use
	%API Utilization	Fraction of API methods that are actually used
[Library,Developers | FAMILIARITY]	Avg. #years of experience	Developers are interviewed or their CVs are consulted
[License,System | COMPATIBILITY]	Expert assessment	Analysis of license terms, *e. g.*, by legal expert

2) Pre-selection: Under ideal circumstances, all libraries should be assessed in full detail – however, we are aware that in practice the sheer number of libraries and the limited resources will make this unrealistic. Therefore, we propose to use the results of the automated analyses as the basis for a pre-selection of libraries: the union of the first N libraries in each category are candidates for detailed inspection and subjected to an expert assessment[4].

B. Expert assessment

Our model guides the expert during the assessment process. The automated analyses have provided the information which can be extracted from the source code. The expert now needs to evaluate the remaining metrics. For this, he or she requires detailed knowledge about the project and its domain. Furthermore, detailed information about the libraries needs to be researched.

In practice, some library characteristics, such as incompatible licenses or security issues might be an exclusion criterium for libraries. If the expert is aware of such criteria, we advise them to assess them for all the libraries of the project. Following the metric evaluation, the expert needs to map the results to the scale $\{low, medium, high\}$. According to the impact relationships in Figure 2, these values map to $\{bad, satisfactory, good\}$.

The goal of our approach is to assess the impact each library has on the activities in our model. Therefore, for each library we count the number of $\{bad, satisfactory, good\}$ scores it has received for each single activity. In the same way, the overall score of a library is computed.

[4]The number N of libraries in the assessment can be determined based on the resources available to conduct the assessment.

C. Report generation

Subsequent to the assessment, a report can be generated from our model. The structure is the following: a chapter is dedicated to each external library. The expert enters the description of the library and fills in the sections generated from the model categories. At the end of each chapter a table summarizes the assessment for the respective library according to the model in Figure 2. At the end of the report, we include an overview which shows the aggregated results for all the libraries in the project. For each library, one row holds the scores per activity in the final table. Also the overall score of the library is reported. Table II shows an example. By offering the detailed chapters with the assessment of all libraries as well as the aggregated view on the system, we cater to the needs of experienced as well as inexperienced recipients, who may be consultants, project managers, or developers. As it highlights potential areas of concern, the report is supposed to serve as basis for reuse and maintenance decisions.

V. TOOL SUPPORT

The assessment model includes five metrics that can be automatically determined by static analyses of the software system to be assessed. Four of these metrics are used in the pre-selection phase of our approach to rank the libraries with regards to their significance to the project. The tool support for the assessment is implemented in Java on top of the

Table II
EXAMPLE FOR ASSESSMENT AGGREGATION

Library	Modify	Understand	Migrate	Protect	Distribute	Overall
Library	G: 2 S: 2 B: 1	G: 2 S: 1 B: 2	G: 1 S: 0 B: 3	G: 0 S: 2 B: 0	G: 0 S: 0 B: 1	G: 5 S: 5 B: 7

Legend: G: # of *good* impacts, S: # of *satisfactory* impacts, B: # of *bad* impacts

open source software quality assessment toolkit ConQAT[5], a modular toolkit for creating quality dashboards which integrate the results of multiple quality analyses. The current implementation is targeted at analyzing the library usage of Java systems but could be adapted to other programming languages with a library reuse concept and for which a parser API in Java is available.

The analysis requires the source and byte code of the project as well as the included libraries as input. The output is a set of HTML and data files showing the metric values for each library in a tabular fashion. The analysis traverses the abstract syntax tree (AST) for each class in the project and determines all method calls to external libraries. For each library, it determines the following five metrics (see also Table I):

- Number of API method calls
- Number of distinct API method calls
- Percentage of affected classes
- Scatteredness of the API
- Percentage of API utilization

The number of total and distinct API method calls as well as the percentage of affected classes are aggregated during the AST-traversal. The scatteredness metric requires more computation: it expresses the degree of distribution of API calls over the system structure. API calls within one package are considered as *local*. We would expect local calls for specific functionality, *e. g.*, calls to networking or image rendering libraries. These would be expected to be concentrated to small parts of the system. Contrarily, libraries providing cross cutting functionality such as logging would be expected to be called from a large portion of the system, therefore exhibiting a high scatteredness value. We compute scatteredness as the sum of the distances between all pairs of package nodes in the package tree with calls to a specific API. The distance of two nodes in the package tree is given by the sum of the distance from each node to their least common ancestor. It is important to note that since the scatteredness metric depends on the system structure (*i. e.*, the depth of the package tree) its values cannot be compared in a meaningful way across different software systems. The percentage of API utilization is computed as fraction between the number of distinct API methods called and the total number of API methods in the library. The complete tool support is available as a ConQAT extension and can be downloaded as a self-contained bundle including ConQAT[6].

[5]http://www.conqat.org/

VI. CASE STUDY

A. Study Goal

To show the applicability of our approach, we performed a case study on a real-world software system of azh Abrechnungs- und IT-Dienstleistungszentrum für Heilberufe GmbH, a customer of CQSE GmbH (see Section II).

B. Analyzed System

The analyzed system is a distributed billing application with a distinct data entry component running on a J2EE application server which is accessed from around 350 fat clients (based on Java Swing). The system's source code comprises about 3.5 MLOC. The system's files include 87 Java Archive Files (JARs).

C. Study Procedure

We executed our assessment approach (see Section IV) on the study object and recorded our observations during the process. We presented our results to the stakeholders in the company and qualitatively captured their feedback. For this we used the following guiding questions:

- Does the report contain the *central* libraries?
- Does the assessment conform to the stakeholders' intuition?
- Are important aspects missing in the assessment?
- Were parts of the assessment result surprising?

D. Results and Observations

The pre-selection step revealed that out of the 87 JAR files included by the project files, the system's source code directly calls methods from 47. The extent of entangledness between the system and these libraries differs significantly, as illustrated in Figure 3(a). For some libraries, only one method is called while for others, there are several thousand method calls indicating the difference in importance for the project. Also the degree of scatteredness varies significantly, as shown in Figure 3(b).[7]

[6]http://www4.in.tum.de/~ccsm/library-usage-assessment/

[7]Note that the long tail of libraries with only one method call or scatteredness of 1 or 0 is represented by the blanks in Figures 3(a) and 3(b), as they are not visible due to the log-scale.

The pre-selection step to determine the most *important* libraries returned a set of 20 JARs. In two cases we conceptually merged several individual JARs into one logical library as they originated from the same in-house project, which resulted in 10 *logical* libraries for further analysis. We then determined for each library all metrics of our assessment model (see Table I) and evaluated each library. The aggregated result of the library usage assessment for the studied software system is shown in Table III.

After a feedback cycle with the CQSE consultant we learned that Spring would have been expected as a central library but was missing in our results. The reason for this is that Spring is a dependency injection framework, in which the framework code mainly calls the user's code and thus the source code does not contain many API method calls. This is a threat to the internal validity of our analysis (see Section VIII). After the consultant's feedback, we added Spring to the list of libraries and performed a detailed assessment for it. The result is highlighted in grey in Table III.

The complete assessment process for a single library took around 20-60 minutes. Without pre-selection, even in the best case, the assessment of all 87 included libraries would have required 29 person hours. In contrast, the assessment of the *significant* libraries identified by pre-selection decreased the effort to approximately 5 person hours.

E. Interpretation

The results in Table III show a mature style of external library usage: most assessed libraries include significantly more positive than satisfactory or negative scores. This is, amongst others, due to a good choice of libraries as no immature or insufficiently supported libraries are included. Another reason for the good results is the high familiarity of the developers with the libraries that helps to overcome potential problems. Also, the support for most libraries is excellent, which positively influences most activities.

The libraries with the best scores are JFormDesigner and Jasper-reports, with 22 and 20 good scores respectively. The libraries introducing most risk are Drools (9×bad), Jai_codec (9×bad) and azh-library1 (10×bad). Depending on the activities, there is a lot of variation concerning the scores of the libraries. The libraries score very heterogeneously for *Modify*, *Understand* and *Migrate*. Contrarily, *Protect* and *Distribute* are well supported. The only exception is Jasper-report as the vendor's commitment to an open-source distribution model is not entirely clear.

F. Stakeholder Feedback

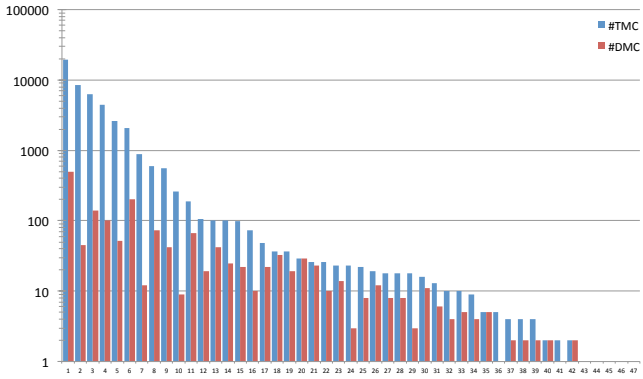
The consultant of the CQSE reported that he perceives the automated pre-selection process as highly beneficial because it allows a selection of *central* libraries based on quantitative data. As not all libraries can be assessed in a typical audit this selection is essential; however, up to now, it was often based on *educated guesses* and opinions of the architects and developers. The metric-based pre-selection helps to make the selection more objective. This is particularly important as quality audits often stir emotions and, hence, all aspects of the audit must withstand fierce criticism and may not appear to be subjective.

Regarding the questions formulated in Section VI-C, the CQSE consultant replied that the report contains the central libraries with the exception of the Spring framework, as discussed above. The results conform to the consultant's intuition as he did not expect the system to depend on inadequate libraries. However, one aspect missing in the assessment is a peculiarity of the Drools library: Drools is not only a library but also defines its own rule description language. Central parts of the assessed system are implemented in this language. Hence, Drools must be considered more central than *normal* libraries. This is not directly reflected in the assessment. The analysis returned some surprising results w.r.t. to the centrality of the libraries. For example, the imaging library JAI was not considered by the consultant before it was pointed out by the analysis.

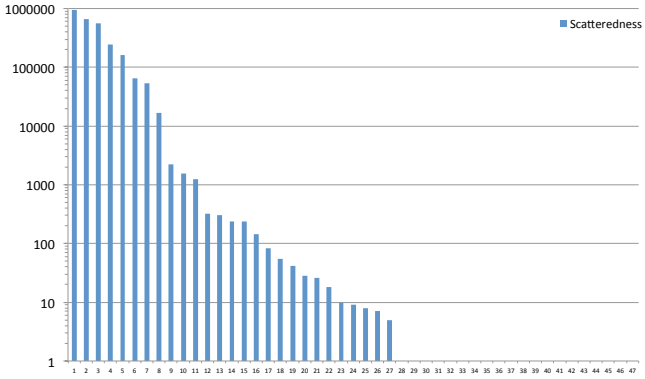
The software engineer of azh reported that the most important libraries of the project were selected. In general, the results conformed to the developer's intuition. However, in some cases more details of the assessment would be helpful. According to the engineer, all important issues were evaluated in our model. In addition, explicit statements about the future viability of the libraries or alternatives to the previously used libraries would be interesting. Parts of the assessment result were surprising, in particular that the libraries JFormDesigner and azh-library2 were among the most central libraries. Overall, the positive outcome of the assessment corresponded to the engineer's perception of the state of library usage on the project.

VII. Discussion

The system analyzed in the case study employs a considerable number of software libraries. While some of them play a central role and thus have a significant impact on the maintenance and further evolution, others are of lesser importance. Our ranking step in the assessment process was able to identify these *important* libraries, allowing to focus further investigation on their usage adequacy. Supported by the assessment model, we identified a number of metrics for each of the identified libraries leading to a comprehensive usage assessment per library. The assessment shows which development activities are influenced and thus allows to identify problematic library usage at a glance. However, proposing *one-size-fits-all* solutions for addressing the issues uncovered by the analysis does not seem adequate; the possible actions are highly dependent on the project context and the maintenance strategies. Our findings rather serve the purpose to create awareness and provide the information for decision making.



(a) The distribution of total vs. distinct method calls.



(b) The distribution of scatteredness.

Figure 3. Result distribution for total and distinct method calls and scatteredness for each JAR file used by the system.

Table III
AGGREGATED VIEW OF THE ASSESSMENT RESULTS

Library	Modify	Understand	Migrate	Protect	Distribute	Overall
Castor	G:2 S:2 B:1	G:3 S:3 B:1	G:2 S:3 B:1	G:5 S:2 B:1	G:2 S:0 B:0	G:14 S:10 B:4
Log4j	G:2 S:1 B:2	G:4 S:1 B:2	G:3 S:1 B:2	G:6 S:0 B:2	G:2 S:0 B:0	G:17 S:3 B:8
Jasper-reports	G:4 S:0 B:1	G:4 S:1 B:2	G:4 S:0 B:2	G:7 S:1 B:0	G:1 S:1 B:0	G:20 S:3 B:5
Drools	G:2 S:1 B:2	G:3 S:1 B:3	G:2 S:1 B:3	G:4 S:3 B:1	G:2 S:0 B:0	G:13 S:6 B:9
azh-library1	G:2 S:1 B:2	G:2 S:2 B:3	G:2 S:1 B:3	G:4 S:2 B:2	G:2 S:0 B:0	G:12 S:6 B:10
azh-library2	G:4 S:0 B:1	G:3 S:0 B:4	G:3 S:0 B:3	G:5 S:2 B:1	G:2 S:0 B:0	G:17 S:2 B:9
Quartz	G:1 S:2 B:2	G:3 S:3 B:1	G:2 S:3 B:1	G:3 S:4 B:1	G:2 S:0 B:0	G:11 S:12 B:5
JForm-Designer	G:4 S:0 B:1	G:5 S:0 B:2	G:4 S:0 B:2	G:7 S:0 B:1	G:2 S:0 B:0	G:22 S:0 B:6
Jai_codec	G:2 S:0 B:3	G:4 S:1 B:2	G:3 S:1 B:2	G:4 S:2 B:2	G:2 S:0 B:0	G:15 S:4 B:9
Ant	G:2 S:2 B:1	G:4 S:1 B:2	G:3 S:1 B:2	G:6 S:1 B:1	G:2 S:0 B:0	G:17 S:5 B:6
Spring	G:3 S:1 B:1	G:3 S:1 B:3	G:3 S:1 B:2	G:5 S:3 B:0	G:2 S:0 B:0	G:16 S:6 B:6

Legend: G: # of *good* impacts, S: # of *satisfactory* impacts, B: # of *bad* impacts

An interesting question is what is considered as an *external* library. For instance a library produced by a different department within the same company may be either considered internal or external. Another central issue is what entity is considered a library. The technical structuring imposed by JAR files may not map to what is considered a logical reusable library. More relevant are factors like provider and release cycle. For an assessment with the proposed model, this has to be defined depending on the specific context in a project. During the discussion of the results, it became apparent that weights for facts would be useful to tailor the model to a specific project context.

VIII. Threats to Validity

A. Internal Validity

The assessment model was created based on CQSE's experience with software quality assessment and software development. We do not know how complete the model is in terms of aspects influencing library usage. However, due to the experience from CQSE GmbH in assessing technology usage in diverse and large industrial projects we are confident that we covered the most central influence factors.

The ranking of the libraries regarding their significance for the including project is based on static analysis metrics taking into account method calls to external libraries. This means that libraries that are indirectly used via other libraries are not considered in the assessment. However, the automated analysis produces a dependency graph of all JAR files including transitively referenced JARs. This allows an assessor to manually identify additional central libraries for detailed assessment.

In addition, reuse can also occur by other means such as subclassing which is typically used in frameworks using a dependency injection mechanism (*e. g.*, Spring). Moreover the analysis cannot detect method calls via the Java Reflection mechanism. In the future, parts of these limitations could be addressed with further static and dynamic analyses.

B. External Validity

Our case study was restricted to a single commercial software system written in Java. We do not know how our findings transfer to software built on other programming ecosystems besides Java. Especially w.r.t. the availability of third-party libraries, we expect major differences.

We are convinced that both the assessment model and the assessment approach have a good applicability to other programming platforms for which a rich variety of reusable libraries exists. More extensive case studies are required to provide evidence this for hypothesis.

IX. Related Work

We relate our approach to the fields of analysis of third-party library usage, architecture analysis and software quality assessment.

A. Analysis of Third-Party Library Usage

In [5], we proposed an approach to determine the degree of dependence between a software project and its third-party libraries in order to support decision making in various use cases during software maintenance. The focus of this previous work was to quantify library reuse while in this paper we present an assessment model which defines what constitutes *adequate* reuse.

Klatt et al. [6] suggested an approach to identify the impact of evolving third-party components on long-living software systems. They use a white-box impact analysis which requires access to the third-party source code and combined it with data from bug trackers and quality analyses on the third-party code. In contrast to our approach, the authors do not provide an explicit model defining the impacts of library usage characteristics to development activities.

Kotonya and Hutchinson [7] suggested an approach that helps developers understanding the impact of change in commercial off-the-shelf (COTS) software components employed in a project. Contrarily to our approach, they rely on a COTS component-oriented development process and focus on the more specific use case of change impact analysis.

Raemaekers et al. [8] proposed an approach to automatically assess the risk imposed by third-party library usage in software projects. They measure the usage frequency of third-party libraries in a corpus of open source and commercial software systems. The risk assessment is based on the assumption that an *uncommon*, *i. e.*, infrequently used, libraries expose a higher risk compared to a library that is frequently employed by software projects. In contrast to our approach, the authors have a very general heuristic for assessing the risk of library usage. We provide a comprehensive model taking multiple factors of libraries and their usage into account.

Lämmel et al. [9] analyzed API usage in 1,476 open source Java projects. They determined the *API usage footprint* of the projects, in terms of the number of included libraries and the number of (distinct) API methods called from the projects' code. Contrarily to our approach, the authors focus on the extent of API usage and do not take into account other characteristics of the library and its usage.

B. Architecture Analysis

Architecture analysis approaches aim at evaluating a software system with regards to its internal structure.

The *software architecture analysis method (SAAM)* proposed by Kazman et al. [10] is an approach for a scenario-based evaluation of software architectures. The method involves describing activities that have to be supported by the software system, prioritizing them, and assessing how well the architecture facilitates them.

The *architecture tradeoff analysis method (ATAM)* [11] is an approach for evaluating a system's architecture with respect to competing quality characteristics (*e. g.*, modifiability

vs. performance). The goal is to to mitigate risks of architectural decisions, ideally early in the development cycle. Potential architectural alternatives are analyzed and a risk mitigation is used to drive refinements of the architectures.

Thus, architecture analysis approaches offer a general framework for the evaluation of principal architectural decisions. In contrast, the proposed library usage assessment approach provides a detailed model of the influence of library usage on maintenance activities.

C. Software Quality Assessment

Software quality assessment methods supported by an explicit quality model, such as Quamoco [12] or Squale [13], use the same principal approach as our method. Attributes of the system and their impacts on quality characteristics are explicitly modeled according to a well-defined meta-model. The model is operationalized in the automated assessment which analyzes a concrete software system with regards to the modeled quality attributes. However, these models do not take into account aspects originating from third-party library usage. In contrast, our approach is specifically targeted at these aspects and is thus complimentary to these approaches.

X. Conclusion and Future Work

Despite their pivotal role in modern software development and their impact on maintenance, third-party libraries tend to be overlooked in quality assessments of software systems. We presented a structured approach to assess the adequacy of library usage in software projects. Based on industrial experience, we provided a lightweight assessment model as well as an assessment process, including tool support and guidance for pre-selecting candidates for inspection. We reported results of a case study applying our approach to an industrial system. The results indicate that our approach gives a comprehensive overview on the external library usage of the analyzed system. It outlines which maintenance activities are supported to which degree by the employed libraries. Furthermore, the semi-automated pre-selection allowed for a significant reduction of the time required by the expert assessment.

Currently an impact between an attribute and an activity is either positive or negative. As a future extension, we plan to extend our meta-model with weights for impacts. This allows for a more fine-grained modeling of the individual impact relationships. Moreover, we want to include custom aggregation functions to allow for a custom weighting in order to emphasize the impact of individual attributes regarding the overall assessment result. A further goal is to increase the automation of the current assessment.

CQSE decided to employ the presented approach in future audits of library reuse. This allows us to collect more data and to evaluate our approach with software from different domains and programming languages.

Acknowledgements

We thank the Google Research Awards Program for supporting our research and the azh GmbH for participating in the study.

References

[1] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the Extent and Nature of Software Reuse in Open Source Java Projects," in *ICSR'11*, 2011.

[2] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J. Girard, "An activity-based quality model for maintainability," in *ICSM'07*, 2007.

[3] B. Kitchenham, S. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation," *Software Engineering, IEEE Transactions on*, vol. 21, no. 12, pp. 929–944, 1995.

[4] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, "Software quality models: Purposes, usage scenarios and requirements," in *WOSQ '09*, 2009.

[5] V. Bauer and L. Heinemann, "Understanding API Usage to Support Informed Decision Making in Software Maintenance," in *CSMR 2012*, 2012.

[6] B. Klatt, Z. Durdik, H. Koziolek, K. Krogmann, J. Stammel, and R. Weiss, "Identify impacts of evolving third party components on long-living software systems," in *CSMR'12*, 2012.

[7] G. Kotonya and J. Hutchinson, "Analysing the impact of change in COTS-based systems," *COTS-Based Software Systems*, pp. 212–222, 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 *CSMR'12*, 2012.

[9] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *SAC'11*, 2011.

[10] R. Kazman, L. Bass, M. Webb, and G. Abowd, "SAAM: A method for analyzing the properties of software architectures," in *ICSE'94*, 1994.

[11] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *ICECCS'98*, 1998.

[12] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit, "The quamoco product quality modelling and assessment approach," in *ICSE'12*, 2012.

[13] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues, "The squale model—A practice-based industrial quality model," in *ICSM'09*, 2009.