On the Extent and Nature of Software Reuse in Open Source Java Projects

Lars Heinemann, Florian Deissenboeck, Mario Gleirscher,
Benjamin Hummel, and Maximilian Irlbeck

Institut für Informatik, Technische Universität München, Germany
{heineman,deissenb,gleirsch,hummelb,irlbeck}@in.tum.de

Abstract. Code repositories on the Internet provide a tremendous amount of freely available open source code that can be reused for building new software. It has been argued that only software reuse can bring the gain of productivity in software construction demanded by the market. However, knowledge about the extent of reuse in software projects is only sparse. To remedy this, we report on an empirical study about software reuse in 20 open source Java projects with a total of 3.3 MLOC. The study investigates (1) whether open source projects reuse third party code and (2) how much white-box and black-box reuse occurs. To answer these questions, we utilize static dependency analysis for quantifying black-box reuse and code clone detection for detecting white-box reuse from a corpus with 6.1 MLOC of reusable Java libraries. Our results indicate that software reuse is common among open source Java projects and that black-box reuse is the predominant form of reuse.

1 Introduction

Software reuse involves the use of existing software artifacts for the construction of new software [9]. Reuse has multiple positive effects on the competitiveness of a development organization. By reusing mature software components, the overall quality of the resulting software product is increased. Moreover, the development costs as well as the time to market are reduced [7, 11]. Finally, maintenance costs are reduced, since maintenance tasks concerning the reused parts are "outsourced" to other organizations. It has even been stated that there are few alternatives to software reuse that are capable of providing the gain of productivity and quality in software projects demanded by the industry [15].

Today, practitioners and researchers alike fret about the failure of reuse in form of a software components subindustry as imagined by McIlroy over 40 years ago [13]. Newer approaches, such as software product lines [2] or the development of product specific modeling languages and code generation [8], typically focus on reuse within a single product family and a single development organization. However, reuse of existing third party code is—from our observation—a common practice in almost all software projects of significant size. Software repositories on the Internet provide a tremendous amount of freely reusable source code, frameworks and libraries for many recurring problems. Popular examples are

K. Schmid (Ed.): ICSR 2011, LNCS 6727, pp. 207–222, 2011.

the frameworks for web applications provided by the Apache Foundation and the Eclipse platform for the development of rich client applications. Due to its ubiquitous availability in software development, the Internet itself has become an interesting reuse repository for software projects [3, 6]. Search engines like Google Code Search[1] provide powerful search capabilities and direct access to millions of source code files written in a multitude of programming languages. Open source software repositories like Sourceforge[2], which currently hosts almost a quarter million projects, offer the possibility for open source software projects to conveniently share their code with a world-wide audience.

Research problem. Despite the widely recognized importance of software reuse and its proven positive effects on quality, productivity and time to market, it remains largely unknown to what extent current software projects make use of the extensive reuse opportunities provided by code repositories on the Internet. Literature is scarce on how much software reuse occurs in software projects. It is also unclear how much code is reused in black-box or white-box fashion. We consider this lack of empirical knowledge about the extent and nature of software reuse in practice problematic and argue that a solid basis of data is required in order to assess the success of software reuse.

Contribution. This paper extends the empirical knowledge about the extent and nature of code reuse in open source projects. Concretely, we present quantitative data on reuse in 20 open source projects that was acquired with different types of static analysis techniques. The data describes the reuse rate of each project and the relation between white-box and black-box reuse. The provided data helps to substantiate the academical discussion about the success or failure of software reuse and supports practitioners by providing them with a benchmark for software reuse in 20 successful open source projects.

2 Terms

This section briefly introduces the fundamental terms this study is based on.

Software reuse. In this paper, we use a rather simple notion of software reuse: software reuse is considered as the utilization of code developed by third parties besides the functionality provided by the operating system and the programming platform.

We distinguish between two reuse strategies, namely *black-box* and *white-box* reuse. Our definitions of these strategies follow the notions from [17].

White-box reuse. We consider the reuse of code to be of the white-box type, if it is incorporated in the project files in source form, *i. e.*, the internals of the reused code are exposed to the developers of the software. This implies that the

[1] `http://www.google.com/codesearch`
[2] `http://sourceforge.net`

code may potentially be modified. The reuse rate for white-box reuse is defined as the ratio between the amount of reused lines of code and the total amount of lines of code (incl. reused source code).

Black-box reuse. We consider the reuse of code to be of the black-box type, if it is incorporated in the project in binary form, *i. e.*, the internals of the reused code are hidden from the developers and maintainers of the software. This implies that the code is reused *as is*, *i. e.*, without modifications. For black-box reuse the reuse rate is given by the ratio between the size of the reused binary code and the size of the binary code of the whole software system (incl. reused binary code).

3 Methodology

This section describes the empirical study that was performed to analyze the extent and nature of software reuse in open source projects.

3.1 Study Design

We use the Goal-Question-Metric template from [20] for defining this study:

> We analyze *open source projects* for the purpose of *understanding the state of the practice in software reuse* with respect to *its extent and nature* from the viewpoint of *the developers and maintainers* in the context of *Java open source software*.

To achieve this, we investigate the following three research questions.

RQ 1 Do open source projects reuse software? The first question of the study asks whether open source projects reuse software at all, according to our definition.

RQ 2 How much white-box reuse occurs? For those projects that do reuse existing software, we ask how much of the code is reused in a white-box fashion as defined in Section 2. We use as metrics the number of copied lines of code from external sources as well as the reuse rate for white-box reuse.

RQ 3 How much black-box reuse occurs? We further ask how much of the code is reused in a black-box fashion according to our definition. For this question we use as metrics the aggregated byte code size of the reused classes from external libraries and the reuse rate for black-box reuse. Although not covered by our definition of software reuse, we separately measure the numbers for black-box reuse of the Java API, since one could argue that this is also a form of software reuse.

3.2 Study Objects

This section describes how we selected the projects that were analyzed in the study and how they were preprocessed in advance to the reuse analyses.

Table 1. The 20 studied Java applications

System	Version	Description	LOC	Size (KB)
Azureus/Vuze	4504	P2P File Sharing Client	786,865	22,761
Buddi	3.4.0.3	Budgeting Program	27,690	1,149
DavMail	3.8.5-1480	Mail Gateway	29,545	932
DrJava	stable-20100913-r5387	Java Programming Env.	160,256	6,199
FreeMind	0.9.0 RC 9	Mind Mapper	71,133	2,352
HSQLDB	1.8.1.3	Relational Database Engine	144,394	2,032
iReport-Designer	3.7.5	Visual Reporting Tool	338,819	10,783
JabRef	2.6	BibTeX Reference Manager	109,373	3,598
JEdit	4.3.2	Text Editor	176,672	4,010
MediathekView	2.2.0	Media Center Management	23,789	933
Mobile Atlas Creator	1.8 beta 2	Atlas Creation Tool	36,701	1,259
OpenProj	1.4	Project Management	151,910	3,885
PDF Split and Merge	0.0.6	PDF Manipulation Tool	411	17
RODIN	2.0 RC 1	Service Development	273,080	8,834
soapUI	3.6	Web Service Testing Tool	238,375	9,712
SQuirreL SQL Client	Snapshot-20100918_1811	Graphical SQL Client	328,156	10,918
subsonic	4.1	Web-based Music Streamer	30,641	1,050
Sweet Home 3D	2.6	Interior Design Application	77,336	3,498
TV-Browser	3.0 RC 1	TV Guide	187,216	6,064
YouTube Downloader	1.9	Video Download Utility	2,969	99
Overall			**3,195,331**	**100,085**

Selection Process. We chose 20 projects from the open source software repository Sourceforge as study objects. Sourceforge is the largest repository of open source applications on the Internet. It currently hosts 240,000 software projects and has 2.6 million users[3].

We used the following procedure for selecting the study objects[4]. We searched for Java projects with the development status *Production/Stable*. We then sorted the resulting list descending by number of weekly downloads. We stepped through the list beginning from the top and selected each project that was a standalone application, purely implemented in Java, based on the Java SE Platform and had a source download. All of the 20 study objects selected by this procedure were among the 50 most downloaded projects. Thereby, we obtained a set of successful projects in terms of user acceptance. The application domains of the projects were diverse and included accounting, file sharing, e-mail, software development and visualization. The size of the downloaded packages (zipped files) had a broad variety, ranging from 40 KB to 53 MB.

Table 1 shows overview information about the study objects. The *LOC* column denotes the total number of lines in Java source files in the downloaded and preprocessed source package as described below. The *Size* column shows the bytecode sizes of the study objects.

Preprocessing. We deleted test code from the projects following a set of simple heuristics (e.g. folders named test/tests). In few cases, we had to remove code that was not compilable. For one project we omitted code that referenced a commercial library.

[3] `http://sourceforge.net/about`

[4] The project selection was performed on October 5th, 2010.

Table 2. The 22 libraries used as potential sources for white-box reuse

Library	Description	Version	LOC
ANTLR	Parser Generator	3.2	66,864
Apache Ant	Build Support	1.8.1	251,315
Apache Commons	Utility Methods	5/Oct/2010	1,221,669
log4j	Logging	1.2.16	68,612
ASM	Byte-Code Analysis	3.3	3,710
Batik	SVG Rendering and Manipulation	1.7	366,507
BCEL	Byte-Code Analysis	5.2	48,166
Eclipse	Rich Platform Framework	3.5	1,404,122
HSQLDB	Database	1.8.1.3	157,935
Jaxen	XML Parsing	1.1.3	48,451
JCommon	Utility Methods	1.0.16	67,807
JDOM	XML Parsing	1.1.1	32,575
Berkeley DB Java Edition	Database	4.0.103	367,715
JFreeChart	Chart Rendering	1.0.13	313,268
JGraphT	Graph Algorithms and Layout	0.8.1	41,887
JUNG	Graph Algorithms and Layout	2.0.1	67,024
Jython	Scripting Language	2.5.1	252,062
Lucene	Text Indexing	3.0.2	274,270
Spring Framework	J2EE Framework	3.0.3	619,334
SVNKit	Subversion Access	1.3.4	178,953
Velocity Engine	Template Engine	1.6.4	70,804
Xerces-J	XML Parsing	2.9.0	226,389
Overall			**6,149,439**

We also added missing libraries that we downloaded separately in order to make the source code compilable. We either obtained the libraries from the binary package of the project or from the library's website. In the latter case we chose the latest version of the library.

3.3 Study Implementation and Execution

This section details how the study was implemented and executed on the study objects. All automated analyses were implemented in Java on top of our open source quality analysis framework ConQAT[5], which provides—among others—clone detection algorithms and basis functionality for static code analysis.

Detecting White-Box Reuse. As white-box reuse involves copying external source code into the project's code, the sources of reuse are not limited to libraries available at compile time, but can virtually span all existing Java source code. The best approximation of *all existing Java source code* is probably provided by the indices of the large code search engines, such as Google Code Search or Koders. Unfortunately, access to these engines is typically limited and does not allow to search for large amounts of code, such as the 3 MLOC of our study objects. Consequently, we only considered a selection of commonly used Java libraries and frameworks as potential sources for white-box reuse. We selected 22 libraries which are commonly reused based on our experience with both own development projects and systems we analyzed during earlier studies. The libraries

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

are listed in Table 2 and comprise more than 6 MLOC. For the sake of presentation, we treated the Apache Commons as a single library, although it consists of 39 individual libraries that are developed and versioned independently. The same holds for Eclipse, where we chose a selection of its plug-ins.

To find potentially copied code, we used our clone detection algorithm presented in [5] to find duplications between the selected libraries and the study objects. We computed all clones consisting of at least 15 statements with normalization of formatting and identifiers (type-2 clones), which allowed us to also find partially copied files (or files which are not fully identical due to further independent evolution), while keeping the rate of false positives low. All clones reported by our tool were also inspected manually, to remove any remaining false positives.

We complemented the clone detection approach by manual inspection of the source code of all study objects. The size of the study objects only allows a very shallow inspection, based on the names of files and directories (which correspond to Java packages). For this we scanned the directory trees of the projects for files residing in separate source folders or in packages that were significantly different from the package names used for the project itself. The files found this way were then inspected and their source identified based on header comments or a web search. Of course this step only can find large scale reuse, where multiple files are copied into a project and the original package names are preserved (which are typically different from the project's package names). However, during this inspection we are not limited to the 22 selected libraries, but potentially can find other reused code as well.

Detecting Black-Box Reuse. The primary way of black-box reuse in Java programs is the inclusion of libraries. Technically, these are Java Archive Files (JAR), which are zipped files containing the byte code of the Java types. Ideally, one would measure the reuse rate based on the source code of the libraries. However, obtaining the source code for such libraries is error-prone as many projects do not document the exact version of the used libraries. In certain cases, the source code of libraries is not available at all. To avoid these problems and prevent measurement inaccuracies, we performed the analysis of black-box reuse directly on the Java byte code stored in the JAR files.

While JAR files are the standard way of packaging reusable functionality in Java, the JAR files themselves are not directly reused. They merely represent a container for Java types (classes, interfaces, enumerations and annotations) that are referenced by other types. Hence, the type is the main entity of reuse in Java[6]. Our black-box reuse analysis determines which types from libraries are referenced from the types of the project code. The dependencies are defined by the Java Constant Pool [12], a part of the Java class file that holds information about all referenced types. References are method calls and all type usages, induced *e. g.*, by local variables or inheritance. Our analysis transitively traverses the

[6] In addition to JAR files, Java provides a *package* concept that resembles a logical modularization concept. Packages, however, cannot directly be reused.

dependency graph, *i. e.*, also those types that are indirectly referenced by reused types are included in the resulting set of reused types. The analysis approach ensures that in contrast to counting the whole library as reused code, only the subset that is actually referenced by the project is considered. The rationale for this is that a project can incorporate a large library but use only a small fraction of it. To quantify black-box reuse, the analysis measures the size of the reused types by computing their aggregated byte code size. The black-box analysis is based on the BCEL library[7] that provides byte code processing functionality.

Our analysis can lead to an overestimation of reuse as we always include whole types although only specific methods of a type may actually be reused. Moreover, a method may reference certain types but the method itself could be unreachable. On the other hand, our approach can lead to an underestimation of reuse as the implementations of interfaces are not considered as reused unless they are discovered on another path of the dependency search. Details regarding this potential error can be found in the section that discusses the threats to validity (Section 6).

Although reuse of the Java API is not covered by our definition of software reuse, we also measured reuse of the Java API, since potential variations in the reuse rates of the Java API are worthwhile to investigate. Since every Java class inherits from `java.lang.Object` and thereby (transitively) references a significant part of the Java API classes, even a trivial Java program exhibits—according to our analysis—a certain amount of black-box reuse. To determine this *baseline*, we performed the analysis for an artificial minimal Java program that only consists of an empty `main` method. This baseline of black-box reuse of the Java API consisted of 2,082 types and accounted for about 5 MB of byte code. We investigated the reason for this rather large baseline and found that `Object` has a reference to `Class` which in turn references `ClassLoader` and `SecurityManager`. These classes belong to the core functionality for running Java applications. Other referenced parts include the Reflection API and the Collection API. Due to the special role of the Java API, we captured the numbers for black-box reuse of the Java API separately. All black-box reuse analyses were performed with a Sun Java Runtime Environment for Linux 64 Bit in version 1.6.0.20.

4 Results

This section contains the results of the study in the order of the research questions.

4.1 RQ 1: Do Open Source Projects Reuse Software?

The reuse analyses revealed that 18 of the 20 projects do reuse software from third parties, *i. e.*, of the analyzed projects 90% reuse code. *HSQLDB* and *YouTube Downloader* were the only projects for which no reuse—neither black-box nor white-box—was found.

[7] http://jakarta.apache.org/bcel

4.2 RQ 2: How Much White-Box Reuse Occurs?

We attempt to answer this question by a combination of automatic techniques (clone detection) and manual inspections. The clone detection between the code of the study objects and the libraries from Table 2 reported 337 clone classes (*i. e.*, groups of clones) with 791 clone instances all together. These numbers only include clones between a study object and one or more libraries; clones within the study objects or the libraries were not considered. As we had *HSQLDB* both in our set of study objects and the libraries used, we discarded all clones between these two.

Manual inspection of these clones led to the observation that, typically, all clones are in just a few of the file pairs which are nearly completely covered by clones. So, the unit of reuse (as far as we found it) is the file/class level; single methods (or sets of methods) were not copied. Most of the copied files where not completely identical. These changes are caused either by minor modifications to the files after copying them to the study objects, or (more likely) due to different versions of the libraries used. As the differences between the files were minor, we counted the entire file as copied if the major part of it was covered by clones.

By manual inspection of the study objects we found entire libraries copied in four of the study objects. These libraries were either less well-known (GNU ritopt), no longer available as individual project (microstar XML parser), or not released as an individual project but rather extracted from another project (OSM JMapViewer). All of these could not be found by the clone detection algorithm, as the corresponding libraries were not part of our original set.

The results for the duplicated code found by clone detection and the code found during manual inspection are summarized in Table 3. The last column gives the overall amount of white-box reused code relative to the project's size

Table 3. Amount of white-box reuse found by clone detection and manual inspection

System	Clone Detection (LOC)	Manual Inspection (LOC)	Overall Percent
Azureus/Vuze	1040	57,086	7.39%
Buddi			—
DavMail			—
DrJava			—
FreeMind			—
HSQLDB			—
iReport-Designer	298		0.09%
JabRef		7,725	7.06%
JEdit	7,261	9,333	9.39%
MediathekView			—
Mobile Atlas Creator		2,577	7.02%
OpenProj	87		0.06%
PDF Split and Merge			—
RODIN	382		0.14%
soapUI	2,120		0.89%
SQuirreL SQL Client			—
subsonic			—
Sweet Home 3D			—
TV-Browser	513		0.27%
YouTube Downloader			—
Overall	11,701	76,721	*n.a.*

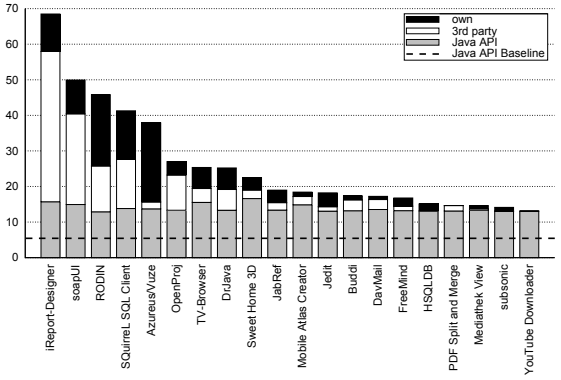

Fig. 1. Absolute bytecode size distribution (MB)

in LOC. For 11 of the 20 study objects no white-box reuse whatsoever could be proven. For another 5 of them, reuse is below 1%. However, there are also 4 projects with white-box reuse in the range of 7% to 10%. The overall LOC numbers shown in the last row indicate that the amount of code that results from copying entire libraries outnumbers by far the code reused by more selective copy&paste.

4.3 RQ 3: How Much Black-Box Reuse Occurs?

Figure 1 illustrates the absolute bytecode size distributions between the project code (own), the reused parts of the libraries (3rd party) and the Java API ordered descending by the total amount of bytecode. The horizontal line indicates the baseline usage of the Java API. The reuse of third party libraries ranged between 0 MB and 42.2 MB. The amount of reuse of the Java API was similar among the analyzed projects and ranged between 12.9 MB and 16.6 MB. The median was 2.4 MB for third party libraries and 13.3 MB for the Java API. The project *iReport-Designer* reused the most functionality in a black-box fashion both from libraries and from the Java API. The project with the smallest extent of black-box reuse was *YouTube Downloader*.

Figure 2 is based on the same data but shows the relative distributions of the bytecode size. The projects are ordered descending by the total amount of relative reuse. The relative reuse from third party libraries was 0% to 61.7% with a median of 11.8%. The relative amount of reused code from the Java API ranged between 23.0% and 99.3% with a median of 73.0%. Overall (third party and Java API combined), the relative amount of reused code ranged between 41.3% and 99.9% with a median of 85.4%. The project *iReport-Designer* had the highest black-box reuse rate. *YouTube Downloader* used the most code from the Java API relative to its own code size. For 19 of the 20 projects, the amount of reused code was larger than the amount of own code. Of the overall amount of reused code in the sample projects, 34% stemmed from third party libraries and 66% from the Java API.

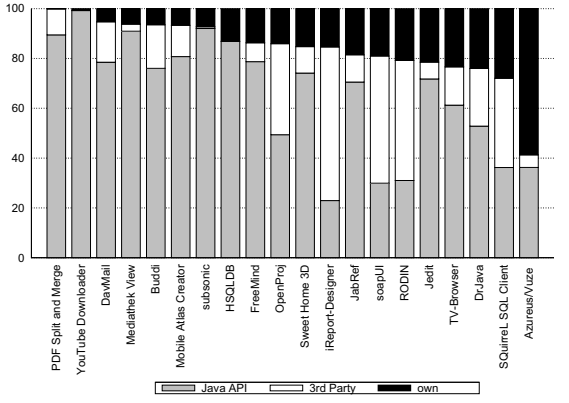

Fig. 2. Relative bytecode size distribution (%)

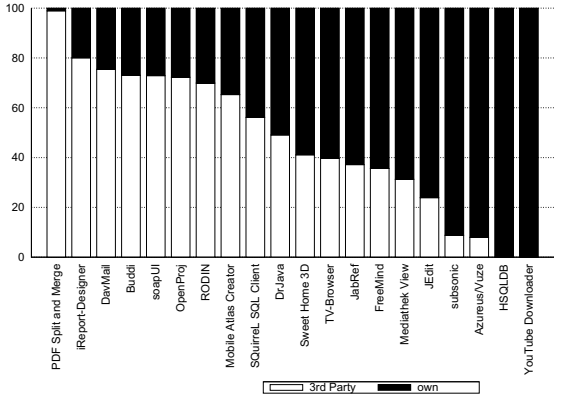

Fig. 3. Relative bytecode size distribution (%) without Java API

Figure 3 illustrates the relative byte code size distributions between the own code and third party libraries, *i. e.*, without considering the Java API as a reused library. The projects are ordered descending by reuse rate. The relative amount of reused library code ranged from 0% to 98.9% with a median of 45.1%. For 9 of the 20 projects the amount of reused code from third party libraries was larger than the amount of own code.

5 Discussion

The data presented in the previous sections lead to interesting insights into the current state of open source Java development, but also open new questions which were not part of our study setup. We discuss both in the following sections.

5.1 Extent of Reuse

Our study reveals that software reuse is common among open source Java projects, with black-box reuse as the predominant form. None of the 20 projects analyzed has less than 40% black-box reuse when including the Java API. Even when not considering the Java API the median reuse rate is still above 40% and only 4 projects are below the 10% threshold. Contrary, white-box reuse is only found in about half of the projects at all and never exceeds 10% of the code.

This difference can probably be explained by the increased maintenance efforts that are commonly associated with white-box reuse as described by Jacobson et al. [7] and Mili et al. [14]. The detailed results of RQ 2 also revealed that larger parts consisting of multiple files were mostly copied if either the originating library was no longer maintained or the files were never released as an individual library. In both cases the project's developers would have to maintain the reused code in any case, which removes the major criticism of white-box reuse.

It also seems that the amount of reused third party libraries seldom exceeds the amount of code reused from the Java API. The only projects for which this is not the case are *iReport-Designer*, *RODIN* and *soapUI*, from which the first two are built upon NetBeans respectively Eclipse, which provide rich platforms on top of the Java API.

Based on our data, it is obvious that the early visions of reusable components that only have to be connected by small amounts of glue code and would lead to reuse rates beyond 90% are not realistic today. On the other hand, the reuse rates we found are high enough to have a significant impact on the development effort. We would expect that reuse of software, as it is also fostered by the open source movement, has a huge contribution to the rich set of applications available today.

5.2 Influence of Project Size on Reuse Rate

The amount of reuse ranges significantly between the different projects. While *PDF Split and Merge* is just a very thin wrapper around existing libraries, there are also large projects which have (relatively) small reuse rates (*e. g.*, less than 10% for *Azureus* without counting the Java API).

Motivated by a study by Lee and Litecky [10], we investigated a possible correlation between code size and reuse rate in our data set. Their study was based on a survey in the domain of commercial Ada development on 73 samples and found a *negative influence* of software size on the rate of reuse. For the reuse rate without the Java API (only third party code) we found a Spearman correlation coefficient of 0.05 with the size of the project's own code (two-tailed p-value: 0.83). Thus, we can infer no dependence between these values. If we use the overall reuse rate (including the Java API), the Spearman coefficient is -0.93 (p-value < 0.0001), which indicates a significant and strong negative correlation. This confirms the results of [10] that project size typically reduces the reuse rate.

5.3 Types of Reused Functionality

It is interesting to investigate what kind of functionality is actually reused by software. Therefore, we tried to categorize all reused libraries into different groups of common functionality. Consequently, we analyzed the purpose of each reused library and divided them into seven categories (*e. g.*, Networking, Text/XML, Rich Cient Platforms or Graphics/UI). To determine to which extent a certain type of functionality is reused we employed our black-box reuse detection algorithm presented in Section 3.3 to calculate the amount of bytecode for each library that is reused inside a project.

We observed that there is no predominant type of reused functionality and that nearly all projects are reusing functionality belonging to more than one category. We believe that there is no significant insight we can report except that reuse seems to be diverse among the categories and is not concentrated on a single purpose.

6 Threats to Validity

This section discusses potential threats to the internal and external validity of the results presented in this paper.

6.1 Internal Validity

The amount of reuse measured fundamentally depends on the definition of *software reuse* and the techniques used to measure it. We discuss possible flaws that can lead to an overestimation of the actual reuse, an underestimation, or otherwise threaten our results.

Overestimation of reuse. The measurement of white-box reuse used the results of a clone detection, which could contain false positives. Thus, not all reported clones indicate actual reuse. To mitigate this, we manually inspected the clones found. Additionally, for both the automatically and manually found duplicates, it is not known whether the code was copied *into* the study objects or rather *from* them. However, all findings were manually verified, for example by checking the header comments, we ensured that the code was actually copied from the library into the study object.

Our estimation of black-box reuse is based on static references in the bytecode. We consider a class as completely reused if it is referenced, which may not be the case. For example, the method holding the reference to another class might never be called. Another possibility would be to use dynamic analysis and execution traces to determine the amount of reused functionality. However, this approach has the disadvantage that only a finite subset of all execution traces could be considered, leading to a potentially large underestimation of reuse.

Underestimation of reuse. The application of clone detection was limited to a fixed set of libraries. Thus, copied code could be missed as the source it was taken from was not included in our comparison set. Additionally, the detector might miss actual clones (low recall) due to weak normalization settings. To adress this, we chose settings that yield higher recall (at the cost of precision). The manual inspection of the study objects' code for further white-box reuse is inherently incomplete; due to the large amounts of code only the most obvious copied parts could be found.

The static analysis used to determine black-box reuse misses certain dependencies, such as method calls performed via Java's reflection mechanism or classes that are loaded based on configuration information. Additionally, our analysis can not penetrate the boundaries created by Java interfaces. The actual implementations used at run-time (and their dependencies) might not be included in our reuse estimate. To mitigate this, one could search for an implementing class and include the first match into the further dependency search and the result set. However, preliminary experiments showed that this approach leads to a large overestimation. For example a command line program that references an interface that is also implemented by a UI class could lead us to the false conclusion that the program reuses UI code.

There are many other forms of software reuse that are not covered by our approach. One example are reusable generators. If a project uses a code generator to generate source code from models, this would not be detected as a form of reuse by our approach. Moreover, there are many other ways in which software components can interact with each other besides use dependencies in the source code. Examples are inter-process communication, web services that utilize other services via SOAP calls, or the integration of a database via an SQL interface.

6.2 External Validity

While we tried to use a comprehensible way of sampling the study objects, it is not clear to what extent they are representative for the class of open source Java programs. First, the choice of Sourceforge as source for the study objects could bias our selection, as a certain kind of open source developers could prefer other project repositories (such as Google Code). Second, we selected the projects from the 50 most downloaded ones, which could bias our results.

As the scope of the study are open source Java programs, transferability of the results to other programming languages or commercially developed software is unclear. Especially the programming language is expected to have a huge impact on reuse, as the availability of both open source and commercial reusable code heavily depends on the language used.

7 Related Work

Software reuse is a research field with an extensive body of literature. An overview of different reuse approaches can be found in the survey from Krueger [9]. In the

following, we focus on empirical work that aims at quantifying the extent of software reuse in real software projects.

In [18], Sojer et al. investigate the usage of existing open source code for the development of new open source software by conducting a survey among 686 open source developers. They analyze the degree of code reuse with respect to developer and project characteristics. They report that software reuse plays an important role in open source development. Their study reveals that a mean of 30% of the implemented functionality in the projects of the survey participants is based on reused code. Since Sojer et al. use a survey to analyze the extent of code reuse, the results may be subject to inaccurate estimates of the respondents. Our approach analyzes the source code of the projects and therefore avoids this potential inaccuracy. Our results are confirmed by their study, since they also report that software reuse is common in open source projects.

Haefliger et al. [4] analyzed code reuse within six open source projects by performing interviews with developers as well as inspecting source code, code modification comments, mailing lists and project web pages. Their study revealed that all sample projects reuse software. Moreover, the authors found that by far the dominant form of reuse within their sample was black-box reuse. In the sample of 6 MLOC, 55 components which in total account for 16.9 MLOC were reused. Of the 6 MLOC, only about 38 kLOC were reused in a white-box fashion. The developers also confirmed that this form of reuse occurs only infrequently and in small quantities. Their study is related to ours, however the granularity for the black-box analysis was different. While they treated whole components as reusable entities, we measured the fraction of the library that is actually used. Since they use code repository commit comments for identifying white-box reuse, their results are sensitive with regards to the accuracy of these comments. In contrast, our method utilizes clone detection and is therefore not dependent on correct commit comments. Their study confirms our finding that black-box is the by far predominant form of reuse.

In [16], Mockus investigates large-scale code reuse in open source projects by identifying components that are reused among several projects. The approach looks for directories in the projects that share a certain fraction of files with equal names. He investigates how much of the files are reused among the sample projects and identify what type of components are reused the most. In the studied projects, about 50% of the files were used in more than one project. Libraries reused in a black-box fashion are not considered by his approach. While Mockus' work quantifies how often code entities are reused, our work quantifies the fraction of reused code compared to the own code within projects. Moreover, reused entities that are smaller than a group of files are not considered. However, their results are in line with our findings regarding the observation that code reuse is commonly practiced in open source projects.

In [10], Lee et al. report on an empirical study that investigates how organizations employ reuse technologies and how different criteria influence the reuse rate in organizations using Ada technologies. They surveyed 500 Ada professionals from the ACM Special Interest Group on Ada with a one-page questionnaire.

The authors determine the amount of reuse with a survey. Therefore their results may be inaccurate due to subjective judgement of the respondents. Again, our approach mitigates this risk by analyzing the source code of the project.

In [19], von Krogh et al. report on an exploratory study that analyzes knowledge reuse in open source software. The authors surveyed the developers of 15 open source projects to find out whether knowledge is reused among the projects and to identify conceptual categories of reuse. They analyze commit comments from the code repository to identify accredited lines of code as a direct form of knowledge reuse. Their study reveals that all the considered projects do reuse software components. Our observation that software reuse is common in open source development is therefore confirmed by their study. Like Haefliger et al., Krogh et al. rely on commit comments of the code repository with the already mentioned potential drawbacks.

Basili et al. [1] investigated the influence of reuse on productivity and quality in object-oriented systems. Within their study, they determine the reuse rate for 8 projects developed by students with a size ranging from about 5 kSLOCs to 14 kSLOCs. While they report reuse rates in a similar range as those from our results, they analyzed rather small programs written by students in the context of the study. In contrast to that, we analyzed open source projects.

8 Conclusions and Future Work

Software reuse, often called the holy grail of software engineering, has certainly not been found in the form of reusable components that simply need to be plugged together. However, our study not only shows that reuse is common in almost all open source Java projects but also that significant amounts of software are reused: Of the analyzed 20 projects 9 projects have reuse rates of more than 50%—even if reuse of the Java API is not considered. Reassuringly, these reuse rates are to a great extent realized through black-box reuse and not by copy&pasting source code.

We conclude that in the world of open-source Java development, high reuse rates are not a theoretical option but are achieved in practice. Especially, the availability of reusable functionality, which is a necessary prerequisite for reuse to occur, is well-established for the Java platform.

As a next step, we plan to extend our studies to other programming ecosystems and other development models. In particular, we are interested in the extent and nature of reuse for projects implemented in legacy languages like COBOL and PL/1 on the one hand and currently hyped languages like Python and Scala on the other hand. Moreover, our future studies will include commercial software systems to investigate to what extent the open-source development model promotes reuse.

Acknowledgment

The authors want to thank Elmar Juergens for inspiring discussions and helpful comments on the paper.

References

1. Basili, V., Briand, L., Melo, W.: How reuse influences productivity in object-oriented systems. Communications of the ACM 39(10), 116 (1996)
2. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns, 6th edn. Addison-Wesley, Reading (2007)
3. Frakes, W., Kang, K.: Software reuse research: Status and future. IEEE Transactions on Software Engineering 31(7), 529–536 (2005)
4. Haefliger, S., Von Krogh, G., Spaeth, S.: Code Reuse in Open Source Software. Management Science 54(1), 180–193 (2008)
5. Hummel, B., Juergens, E., Heinemann, L., Conradt, M.: Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In: ICSM 2010 (2010)
6. Hummel, O., Atkinson, C.: Using the web as a reuse repository. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, pp. 298–311. Springer, Heidelberg (2006)
7. Jacobson, I., Griss, M., Jonsson, P.: Software reuse: architecture, process and organization for business success. Addison-Wesley, Reading (1997)
8. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling. Wiley, Chichester (2008)
9. Krueger, C.: Software reuse. ACM Comput. Surv. 24(2), 131–183 (1992)
10. Lee, N., Litecky, C.: An empirical study of software reuse with special attention to Ada. IEEE Transactions on Software Engineering 23(9), 537–549 (1997)
11. Lim, W.: Effects of reuse on quality, productivity, and economics. IEEE Software 11(5), 23–30 (2002)
12. Lindholm, T., Yellin, F.: Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
13. McIlroy, M., Buxton, J., Naur, P., Randell, B.: Mass produced software components. In: Software Engineering Concepts and Techniques, pp. 88–98 (1969)
14. Mili, H., Mili, A., Yacoub, S., Addy, E.: Reuse-Based Software Engineering: Techniques, Organizations, and Controls. Wiley Interscience, Hoboken (2001)
15. Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. IEEE Transactions on Software Engineering 21(6), 528–562 (1995)
16. Mockus, A.: Large-scale code reuse in open source software. In: FLOSS 2007 (2007)
17. Ravichandran, T., Rothenberger, M.: Software reuse strategies and component markets. Communications of the ACM 46(8), 109–114 (2003)
18. Sojer, M., Henkel, J.: Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. JAIS (to appear, 2011)
19. von Krogh, G., Spaeth, S., Haefliger, S.: Knowledge Reuse in Open Source Software: An Exploratory Study of 15 Open Source Projects. In: HICSS 2005 (2005)
20. Wohlin, C., Runeson, P., Höst, M.: Experimentation in software engineering: An introduction. Kluwer Academic, Dordrecht (2000)