UNIVERSITÄT KOBLENZ · LANDAU

A Large-Scale Analysis of Java API Usage

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Jürgen Starek

Koblenz, im November 2010

Erstgutachter: Prof. Dr. Ralf Lämmel
Institut für Informatik, AG Softwaresprachen
Zweitgutachter: Dipl. Math. Ekaterina Pek
ebd.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

....

(Ort, Datum) (Unterschrift)

Deutsche Zusammenfassung

Die vorliegende Arbeit stellt eine Methode für die korpusbasierte Analyse der Verwendung von Softwarebibliotheken in Java-Programmen vor.

Die meisten größeren Softwareprojekte verwenden Softwarebibliotheken, die als Schnittstelle für den Programmierer sogenannte *APIs* (application programming interfaces) bereitstellen. Um den Umstieg von einer solchen API auf eine andere zu unterstützen, strebt man Werkzeuge zur *automatisierten API-Migration* an. Zur Entwicklung solcher Werkzeuge fehlen allerdings noch Basisdaten. Statistiken und Beobachtungen zur Verwendung von APIs in der Praxis wurden bisher nur mit sehr kleinen Korpora von Projekten und APIs durchgeführt.

Wir stellen in dieser Arbeit daher eine Analysemethode vor, die für Messungen an großen Korpora geeignet ist. Hierzu erzeugen wir zunächst einen Korpus von auf SourceForge gehosteten Open-Source-Projekten sowie einen Korpus von Softwarebibliotheken. In der Folge werden alle Projekte des Korpus compiliert, wobei ein Compiler-Plugin für den javac detaillierte Informationen über jede einzelne Methode liefert, die der Compiler erstellt. Diese Informationen werden in einer Datenbank gespeichert und analysiert.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Questions and Contributions . . . 1
1.3 Acknowledgements . . . 2
1.4 Structure . . . 2

2 Methods **4**
2.1 Overview . . . 4
2.2 Terms and Definitions . . . 6
2.2.1 APIs . . . 6
2.2.2 Features . . . 6
2.2.3 Code Units . . . 6
2.3 Creating the Corpus . . . 7
2.3.1 Selection Criteria . . . 7
2.3.2 Downloading . . . 8
2.3.3 Defining Reference Projects . . . 8
2.3.4 Defining and Detecting APIs . . . 8
2.3.4.1 Analysis and Tagging . . . 9
2.3.5 Gathering Facts about the Java Environment . . . 9
2.3.5.1 Defining Core Java . . . 9
2.3.5.2 Components of a Sun Java SE Installation . . . 9
2.4 Overview of the AST-based Fact Extraction . . . 10
2.4.1 Building and Fact Extraction . . . 10
2.4.2 Automated Data Preparation and Tagging . . . 10
2.5 Overview of the Lexeme-based Fact Extraction . . . 11
2.5.1 Scanning the Source Code . . . 11
2.5.1.1 Import Statements . . . 11
2.5.1.2 Code Size . . . 12
2.5.2 Data Storage . . . 12
2.6 Threats to Validity . . . 12
2.6.1 In Project Selection . . . 12
2.6.2 In API selection and processing . . . 12
2.6.3 In the AST-based Analysis . . . 13
2.6.4 In the Lexeme-based Analysis . . . 13

3 Tools **14**
3.1 The Compiler Plugin for AST-Based Fact Extraction . . . 14
3.1.1 Tests . . . 16
3.2 Scripts for the Lexeme-based Analyses . . . 17
3.3 The Database . . . 17
3.3.1 Contents of the tables . . . 19
3.3.1.1 Data obtained from the compiler plugin . . . 19

3.3.1.2 Data obtained from analyses of binary distributions 22
3.3.1.3 Data obtained from the text-based scanner and the build system 22
3.4 Manual API Tagging 23
3.5 Automatic API Detection 24
3.6 Reference Projects 24
3.6.1 Data Sources 26
3.6.2 Selection Criteria 26
3.6.3 Script-based Tagging 27
3.7 Clone Detection 27
3.7.1 Inclusion of API Code in other APIs 27
3.7.2 Inclusion of API Code in Project Code 28
3.8 Querying and Plotting 30
3.8.1 Analysis Scripts: Sample Workflow 30

4 Results 33
4.1 Measurements of Corpus Size 33
4.2 Sources of Features in Project Code 34
4.3 API Usage 38
4.4 Feature Usage Frequency Distribution 42
4.4.1 In the entire corpus 42
4.4.2 In APIs 44
4.5 Coverage of API Features 45

5 Related Work 48
5.1 Structure of Real-World Programs 48
5.2 Simple Project Statistics 48
5.3 Clone detection 49
5.4 API Usage 49
5.4.1 Theoretical Considerations 49
5.4.2 Mining Frequent Usage Patterns 49

6 Further work 51
6.1 Alternative approaches 51

7 Conclusion 53

1 Introduction

1.1 Motivation

With the increasing complexity of software, the use of software libraries and their respective APIs has become mainstream, especially in object-oriented languages. Java has created a rich ecosystem of APIs that has attracted a lot of work from the software engineering community. With increasing age of both API and project code in the Java environment, it is now becoming increasingly important to consider the challenges posed by changing a project's potentially large code base to use different APIs than before. Anticipation of those challenges has led to work in the field of API migration.

The work presented in this thesis was intended to be a starting point for further work in that area, more specifically in the field of automated API migration. We noticed that, while there is a broad selection of work on the structure of programs and on automated extraction of usage patterns from code corpora, there are few works that analyze the API usage of Java programs in the wild. We concentrate on providing answers to basic questions about API usage, most importantly, how widespread the use of APIs among a large set of open source programs is. Additionally, we examine the coverage of API features and possible correlations between different basic software metrics and API usage. In doing so, we hope to provide a set of methods and basic data for further work in this area.

1.2 Research Questions and Contributions

This thesis presents an analysis of API usage in a large corpus of Java software retrieved from the open source repositories hosted at SourceForge. There have been few works on API usage analysis, most previous work focusing on analyzing project structure (see section 5.1), repository metadata (see section 5.2) or usage pattern mining (see section 5.4.2). Because of that, one of our initial tasks was to find methods and tools suitable for conducting such an analysis. This part of the work was driven by the following questions:

- How can a large quantity of programs and APIs be made available for an automated code analysis? This includes both the question of data sources and methods for making their data available.
- Which methods are suitable for a pragmatic, large-scale analysis of API usage in such a corpus?
- Can automation help in making results reproducible and in easily incorporating new material in the corpus?

In order to put the measurements from this project into perspective, we had to quantify the importance of API usage in real-world programming. We did this to obtain a point of reference against which we could compare further results. Additionally, we needed to know if this quantity does somehow correlate with the maturing of the projects' code. The questions answered by these measurements are:

- What part of Java software is made up by features from APIs?
- Does this percentage vary with continued development of the programs, and if so, how do project maturity and API usage correlate?
- Are the APIs that are delivered with Java itself used more often that third-party APIs?

While APIs generally concentrate on providing features from a well-defined problem domain, it is unclear how well a given API covers the problems in such a domain. The fact that there are usually several APIs from different development teams that cover the same domain might hint at a low coverage, with many special features left for “competing” implementations. While we can not determine the mechanisms and politics behind feature distribution in APIs, we may measure the amount of different features that are used throughout our corpus:

- How thoroughly are APIs used, and are there API features that are used markedly more often than others?

In [Vel05], Veldhuizen argues that the amount of re-used code in projects depends on the problem domain, contrary to views that expect reuse to increase with further development of the software and increasing code size. From this, we derive further questions:

- Is API usage more prevalent in large projects compared to small ones?
- If so, are the number of used APIs or the number of different API features used correlated with project size?

When, during the work leading up to this thesis, we discovered that many programmers shipped APIs by including verbatim copies of (parts of) API source code into their programs, we needed to deal with that phenomenon. Two new questions arose:

- How can this type of code clones, i.e. API source code being copied into a program's source code, be detected?
- How often are APIs distributed in this way?

As by-products of these analyses, we repeat some measurements of basic structural properties of Java software that were done, amonst others, by Baxter et al. [BFN+06] or Collberg et al. [CMS07].

1.3 Acknowledgements

I would like to thank my advisor, Prof. Dr. Ralf Lämmel, for offering me this interesting topic. In the course of this work, he provided me with many helpful ideas, suggestions and encouragement.

Also, I would like to thank Ruwen Hahn for the implementation of the compiler plugin, as well as for his assistance with writing parts of the analysis software and with the administration of our workstations.

1.4 Structure

Having outlined our basic research questions above, we discuss the methods applied in our analysis and the rationale behind them in chapter 2.

After providing definitions for some of the most important terms used in the remainder of the text, we discuss the sequence of analysis steps and present the two main analysis tools, a fact extractor that gathers information from the abstract syntax tree of a compiled program and a

lexeme-based fact extractor following a more conventional text analysis approach. A discussion of possible sources of systematic errors concludes the chapter.

After this discussion of analysis methods, chapter 3 focuses on a discussion of the software tools we developed. Both the two fact extractors mentioned above and several auxiliary tools are described in detail. The chapter also describes the design of the relational database we used to store measurements and metadata.

The results of the measurements are presented in chapter 4. Graphs and a short discussion are provided for each measurement, along with attempts at an interpretation.

Chapter 5 gives an overview of related work, and chapter 6 discusses possible ways for continuing the work started with this project and improving its methodology.

Chapter 7, summarizing results and perspectives for future work, concludes the thesis.

2 Methods

2.1 Overview

There are several possible approaches to studying API usage empirically. One would be to hand-pick a corpus of representative projects and APIs and analyzing them one after another. However, we decided not to pursue this approach because we could not find universal and useful criteria for picking either APIs or projects – as we aimed for a large corpus, “soft”, informal criteria like “include only projects with a mature codebase” that require human interaction in the selection process were deemed inadequate. There is no good existing theory of API usage that we could have adopted here. Instead, we chose a brute-force approach for the creation of the corpus.

Figure 2.1 gives a simplified overview over the processes that were used for the analyses presented in this thesis: Initially, we obtained a list of the projects that would form our corpus with the help of SourceForge’s search function. The corpus was then created by fetching each of these projects’ source code from the SourceForge SVN repositories. This process is discussed in greater detail in section 2.3.

The corpus data were then processed by the two main analysis methods. The first was to build the projects with an instrumented compiler that gathers information from the programs’ abstract syntax trees (ASTs) and is described in section 2.4, the second was to apply a simple text-based scanner, which is discussed in section 2.5.

Starting with the list of selected projects, all data used in the analyses (except for the source code itself) was kept in a relational database that stores information about the projects, their metadata, and all features and all APIs that the compiler encounters. For each project that is built successfully, we store the set of all feature references that the compiler accessed, with detailed type information. Our analyses do not, however, store control flow- or data flow-based information about the sequence of those feature references. The database structure is discussed in section 3.3.

The selection of the APIs we analyze was driven mainly by data from the corpus. The output from an initial attempt at compiling the entire corpus was fed into a set of analysis scripts, one of which generated a list of the packages that were most frequently reported as missing by the compiler. We then downloaded the APIs that provide the most frequently missed packages, forming a repository of binary API distributions. These API distributions were then processed as described in section 3.4, which made them available in the build environment and gave us detailed information about them. Afterwards, we started another build process attempting to successfully compile the projects that failed in the last run. This was iterated until more than half of all API features we observed in the corpus were provided by APIs which we processed in this manner.

Finally, a set of analysis scripts produced the graphs and tables presented in chapter 4 from the data gathered by the AST-based and the text-based analysis.

Figure 2.1: Simplified overview of the analysis process

2.2 Terms and Definitions

2.2.1 APIs

In the scope of this project, we had to agree on a definition of an "API" that is as precise and comprehensive as it is pragmatic. We intend to examine the use of *software libraries*, i.e. pieces of code that provide often-needed functions that are likely to lend themselves well to re-use, as well as a public *application programming interface* or *API*. Programmers gain access to the library's functions by using that API.

In the remainder of the text, we will drop (as in common parlance) the distinction between the library itself and its API, because using the library functions implies using the API.

For the practical part of this work, we needed to define the relationship between APIs and Java's language features.

Because many APIs are organized into a single Java package, it is tempting to try to define APIs as sets of features with a common package name. Indeed, for some well known APIs such as Java's Reflection API, which is located in the `java.lang.reflect` package, this is sufficient. However, a part of Reflection's functionality requires `java.lang.Class`, which is from a different package. So this definition is not sufficient even for such simple cases.

To remedy this, one may try to use package trees as the basis for a definition, i.e. a set of packages with a common name prefix. The features of dom4j, for example, are all organized into subpackages of `org.dom4j.*`. However, this definition fails to adress the structure of APIs such as JUnit that use different prefixes (`org.junit.*` and `junit.*`, respectively).

So one might attempt to define APIs in a more generalized manner as sets of packages. Unfortunately, while this is an adequate definition for almost all cases, one of the most often used APIs, namely Java's Collections API as described in [Sun], is organized differently. The Collections classes are spread out widely across different packages. Hence, we were forced to treat APIs as *named collections of types*. In most cases, though, a mapping from API names to package name prefixes can be used as a shortcut.

We assign each API to a *domain*. Domains are intended to represent the main usage areas of an API; the junit framework, for example, might be added to the "Testing" domain. Domain names are free-form, and the list of domains was created ad-hoc during the API tagging process.

2.2.2 Features

We aim for a detailed view of API usage which is based on the concept of a *feature*. In the context of this work, the term includes instance methods, static methods and constructors. We ignore fields (which would normally also be counted as features) because APIs generally do not expose them, and in those rare cases where they are exposed, fields are scarcely used independently from methods.

A feature is *provided* by a type if the type's public interface includes the feature. It is *used* or *referenced* if the method or constructor is called. While the provider type for constructors is immediately obvious (namely, the class that the constructor belongs to), we approximate the provider type for methods by the static bound given by Java's type system.

2.2.3 Code Units

We measure API usage for some given *code units*. A code unit may be an entire project, a package, a class or a method. This distinction allows us to perform measurements at different detail levels, which are given for all measurements where they are non-obvious.

API usage is compared to usage of features from *client code* (or *project code*) and *Core Java*. The term client or project code summarizes all features from SourceForge projects that are not themselves software libraries. Our corpus of compiled projects provides all project code we analyzed, and there are no software libraries present in that corpus. Client code is split into *source packages* containing *source types*. These names hint at the different format of project and API code: The former is available as source code in our corpus, whereas the latter is available as pre-packaged JAR archives for use by our build system.

Core Java is a term coined by Sun Microsystems, the original source of the Java programming environment, and refers to a subset of features from a Java installation that can be expected to be included in any Java distribution, regardless of which vendor created the distribution. For example, Core Java features can be expected to be present in both a Sun JDK and an IcedTea-based OpenJDK distribution. A more detailed discussion is provided in section 2.3.5.1.

2.3 Creating the Corpus

Initially, we had to locate, download and store source code for all Java projects that were to form our corpus.

We did not intend to restrict the project corpus to any application domain, but aimed for a large and diverse selection of projects.

As we tried to use as much automation as possible in the creation of the corpus, we needed to find a large data source that uses a homogenuous, predictable naming scheme for its downloads. We chose *SourceForge.net*[1], a large hosting provider for open source projects. SourceForge also offers the additional benefits of allowing direct access to revision control system repositories and providing some meta-information about each project, which lends itself well to automated analyses.

SourceForge does, however, host projects created in many different programming languages and for various different environments. We had to employ some selection criteria in selecting the projects for this study.

2.3.1 Selection Criteria

Criterion: all Java projects

As the first step of the creation of the corpus, we created a list of all Java projects hosted on Sourceforge. This was done by HTML-scraping Sourceforge's search result pages, using simple regular expression matching. This search was conducted in September 2008, and at that time, it yielded 33550 projects.

Criterion: projects accessible through Subversion

For downloading the actual source code, we limited ourselves to projects that offer publicly accessible Subversion repositories on Sourceforge. The reasons for this restriction is that we expect this to be the main way in which projects distribute their source code (as opposed to offering it in the downloadable “builds” available for some projects, which usually consist of ready-to-run binary program files). CVS, Mercurial and git repositories were not downloaded, however, and it will probably be worthwile to add support for downloading from those sources in order to increase corpus size and diversity.

Applying this restriction left 8578 projects.

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

Criterion: automatically buildable

In order to automate the build process (and, accordingly, data acquisition by the instrumented compiler, see section 3.1), we further restricted our corpus to contain only those projects that use Apache ANT build files. The advantage of relying on ANT files is that they offer a simple way for invoking our analysis tools. Besides, relying on ANT relieves us of the task of analysing the structure of projects to find starting points from other build systems (like makefiles, batch / shell scripts etc.). Instead, we may rely on a standardized build process that can be automated easily. Moreover, ANT provides a simple way to check for a successful build.

Because of that, we omitted support for makefiles and projects that do not use any build system. Again, future improvements to our system should include support for these environments in order to increase corpus size and diversity.

4223 projects matched this criterion.

2.3.2 Downloading

After applying the aforementioned restrictions, we extracted the unique SourceForge-internal project numbers from the search results. Using a standard Subversion client, the source code for all selected projects was downloaded, using the project numbers for creating the repository URLs. Just as the search page analysis script, the script used for automatic downloading used frequent pauses of several seconds in order to lessen the impact of our large download on SourceForge's servers.

In the end, the corpus spanned 138 GB, which, however, includes images, documentation files and everything else that developers chose to include in their Subversion repositories. The `.java` files themselves only take up about 17 GB of space.

Later analyses revealed that the source files in the corpus together represent about 264536500 lines of code, ignoring comments, or 377640164 total lines of code.

2.3.3 Defining Reference Projects

The quality of the software in SourceForge's repositories varies greatly. In our corpus, we see tiny projects that seem to have been abandoned after a few weeks as well as large, active projects that have been under development for a long time.

Code quality is hard to measure, as for such a diverse corpus, most of the usual metrics are not suitable. Because we assumed that unfinished projects would not use APIs in an efficient and well thought-out manner, we aimed to define a subset of the projects in the corpus as a control group of stable, working projects.

However, because of the large corpus size, a manual assessment of the projects was not feasible in the given timeframe. We use an automated selection process instead. This process is based on the metadata available through the SourceForge website, relying mainly on the projects' maturity rating. The process is described in detail in 3.6.

2.3.4 Defining and Detecting APIs

As stated in section 2.2.1, we define APIs as sets of types. In order to provide projects with the necessary APIs to compile them successfully, we download the binary distributions of APIs (which, generally, are JAR or ZIP archives containing compiled classfiles). From these binaries, we extract a list of all features they contain and store information about each feature in the database. The binary distributions are then made available to the build process by extracting them to a dedicated library directory.

However, many authors distribute, inside one single downloadable archive, both their own work (the API we are interested in) and third-party APIs or sample programs. One typical exaple is the unit testing framework *JUnit*, which delivers, along with the junit.* and org.junit.* packages that make up the framework itself, the *Hamcrest* library in the org.hamcrest.* packages. Hamcrest ist developed and hosted independently, at `http://code.google.com/p/hamcrest/`.

Because of this, the process of adding a new API to the database was refined beyond simply tagging a set of features as belonging to a certain API.

2.3.4.1 Analysis and Tagging

After downloading the distribution package, an analysis tool (which is described in greater detail in section 3.4) is used to pre-process the API distribution. For each file in a binary API distribution, the tool asks the user for the name of the API, one or more *package prefixes* and, for each jar file contained in the distribution archive, whether or not to include its features in the database. This allows exclusion of third-party code or examples.

The package prefix is the part of the package names that all features in an API have in common. Because of the aforementioned problems in separating API and example code from an archive, this string is used for a more precise separation of the API's features: Only those that originate from a package starting with this package prefix are considered to be part of the API in question.

After this information has been gathered, the API's features are all entered in the database, and those matching the package prefix are marked as belonging to the API.

2.3.5 Gathering Facts about the Java Environment

In preparing the compile runs, some metadata about the build environment was gathered and stored in the database. Before evaluating how "the Java classes" or "the Java APIs" are used, we needed to define the set of classes to be considered as such. According to the following definition, a list of features delivered with a standard Java package was created and stored in the database.

2.3.5.1 Defining Core Java

Sun themselves define a set of classes called *Core Java*, which, although they span APIs as different as Swing and W3C's DOM API, can be thought to form one core API that is accessible to every program running on a Sun JRE.

The Sun Java documentation is not entirely unambiguous with regard to the definition of the core API. Sun states in [JDK] that the archive rt.jar holds "the runtime classes that comprise the Java platform's core API". However, when comparing the contents of this archive to the Java API specification [Javb], it is obvious that it contains many more packages than are mentioned in the API specification.

On the Java website, Sun states that "The java.*, javax.* and org.* packages documented in the Java 2 Platform Standard Edition API Specification make up the official, supported, public interface" and that "The sun.* packages are not part of the supported, public interface" [Javc]. This is the definition used in this analysis: The classes from the java.*, javax.* and org.* packages that are shipped in the rt.jar of the Java SE 1.6.10 distribution form our list of Core Java Classes.

2.3.5.2 Components of a Sun Java SE Installation

Both the Java Development Kit (JDK) and the Java Runtime Environment (JRE) contain several JAR archives, which hold all classes and interfaces made available by Sun. Core Java, as defined

above, is a subset of these classes, but vendors (Sun as well as others) generally extend their Java distributions well beyond that. The following JAR files are present in Sun's 1.6.x series JDK distributions:

rt.jar contains the so-called *bootstrap* classes, which are loaded by the bootstrap class loader [Cla], and some implementation-specific classes.

charsets.jar contains classes for converting between different character representation systems

tools.jar is a Sun-specific archive containing support classes for auxiliary programs in the JDK. These classes, according to Sun, do not belong to the core API.

dt.jar is a set of auxiliary classes, intended to be used by IDEs for the display of Java components at design time. As these classes are not intended to be used directly by the developer, they are not included in this analysis.

localedata.jar contains data for internationalizing text output. Although this archive is located in the $JAVA_HOME/jre/lib/ext directory, it is not considered a part of Core Java.

In order to create a list of all the Core Java features, we used a small Java program to recurse through all JAR files, extract all entries from those archives and write them into the database.

As the JRE is contained in the JDK distribution, we do not need to treat the two separately.

2.4 Overview of the AST-based Fact Extraction

2.4.1 Building and Fact Extraction

The projects were built using a script that uses the information given in the projects' ANT files. It spreads the load across several processes to allow parallelizing the build. The available processor time for each of these build processes is artificially limited to 25 minutes in order to keep projects with programming errors from getting the build machine stuck in endless loops.

Initially, the software environment for building the projects consisted only of a Java Standard Edition installation, version 1.6.10. Many projects failed to build at this stage because of missing packages, so that in subsequent builds, the complete binary distributions of the manually tagged APIs (see chapter 3.4) were made available on the build path. Providing these APIs allowed approximately 200 further projects to build successfully, resulting in a total of 1476 built and analyzed projects that provide the base data for the following analyses. Building took less than a week on a workstation using an AMD Phenom II 940 (3 GHz, four cores) processor.

These automated builds were done using a compiler plugin that instruments the stock javac from our JSE installation. This plugin, which is described in detail in chapter 3.1, provides detailed information about every feature that is compiled.

This information forms the bulk of the data used in our analyses. These build runs provided us with a list of all feature references in all compiled projects, including the class, package and project (or API) the feature was found in.

2.4.2 Automated Data Preparation and Tagging

After collecting the data provided by the compiler plugin from the build runs and the analysis of the JRE, some scripts are used to derive information about the static relationships between selected components. This information is, again, stored in the database.

First, a script tags all packages and features that are part of Java itself (as determined by the methods discussed in section 2.3.5.2) as having been delivered with the respective Java SE

distribution. It also tags the packages that form the Core Java feature set (as discussed in 2.3.5.1) as such.

A second script then applies the API detection algorithm, as described in section 3.5, to the database. The usage pattern of all packages is examined, and those which are likely parts of an API are flagged as such.

Thirdly, the results from the manual API analysis (see section 3.4) are added to the database. A new table is created for storing the analysed APIs, all their features, classes and packages are stored in the database, and those API features that were used in the corpus are tagged as belonging to a known API.

The last script uses meta-information from the Subversion repositories and the project websites to identify and tag a set of mature, well-developed reference projects. This script is described in greater detail in section 3.6.

2.5 Overview of the Lexeme-based Fact Extraction

The plugin-based analysis as described above has one main disadvantage: It can only be applied to projects that can be built using our automated ant-based process. But a lot of projects we downloaded failed to build for various reasons. Apart from programming errors, one of the most common problems is caused by missing packages. This type of error, however, is far less likely to occur for projects using only Core APIs and APIs from the set that we tagged manually, as all those are available in the build environment. The results we obtained from the plugin-based analyis are, thus, skewed, showing a preference for these APIs over those that are used more rarely.

2.5.1 Scanning the Source Code

In order to assess the severity of this problem, we implemented a simple source-based analysis. A script recurses through the source code of all downloaded projects, extracting the packages that are included in each `.java` file.

2.5.1.1 Import Statements

The Java Language Specification defines four kinds of import statements:

- single-type-import declarations
- type-import-on-demand declarations
- single static import declarations
- static-import-on-demand declarations

A single-import declaration gives the full name of a class, and makes all its accessible types available in the scope of the declaration. An on-demand import declaration, which ends with an asterisk after the package name, imports all accessible types from the given package, while a static import declaration imports the static members of the given package and makes them available statically. Our scanner discerns those four cases.

If, however, member functions are called using the fully qualified names of their enclosing classes, as specified in § 6.7 of the Java Language Specification , we will miss these references by using only import statements.

2.5.1.2 Code Size

While analyzing the source files, both the number of lines in each file and the number of non-comment lines are stored in the database. Both values can serve as an estimate of the lines of code in a file and, hence, as a proxy for the complexity of the code units.

Comments are excluded by a simple algorithm that discards lines which either start with double slashes or which lie between the delimiters of a *traditional comment*, i.e. between /* and */.

Empty lines are counted as valid lines of code for both measurements.

2.5.2 Data Storage

The scanner data is stored in two tables in the database. One holds references to each file, the project it was downloaded from and both its total line count and the lines of code (see above). The other table is used to store the import statements as encountered in each file, with flags for on-demand-imports and static imports. From this representation, all imports of each project can be reconstructed for analysis.

2.6 Threats to Validity

2.6.1 In Project Selection

Howison ([HC04]) points out that "SourceForge has become the 'repository of record' for the FLOSS community, yet for important projects it is not the 'repository of use'". Many large and high-profile OSS projects are not developed on SourceForge's repositories, which means that any analysis that only relies on SourceForge is missing a lot of high quality code. Howison cites the vim editor as an example, a project for which development happens in a project-managed repository and not on SourceForge.net. vim is written in C, but for our Java-based analysis, the Eclipse IDE can serve as a similar example: The project is stable, large and mature, and its code has been reviewed by many parties. It is, however, developed in its own repositories[2] and (with the exception of supplementary projects) invisible for scanners that are restricted to the sourceforge.net domain.

Besides, any selection strategy similar to ours (compare [HK05]) makes systematic errors in leaving out projects that are hosted using unsupported revision control systems, on third-party repositories or that are simply not classified correctly (e.g. with an incorrect programming language tag) at SourceForge.

2.6.2 In API selection and processing

Because of the manual steps involved in the processing of binary API distributions (see 2.3.4), several problems arose that require further work in order to use the analysis method described in this thesis efficiently. Most importantly, the lack of an automatically repeatable exclusion process lead to an inability to correct an issue with the database not storing information about access modifiers (public, private etc.) for features from JAR archives without repeating the entire selection and build process. For the API coverage graphs in 4.13, the base set is still the entire set of API features including private ones.

While this is not a systematic problem of the analysis approach we describe, this problem should be addressed in future implementations.

[2] `http://dev.eclipse.org/viewcvs/index.cgi`

2.6.3 In the AST-based Analysis

The AST analysis by the compiler plugin can obviously only scan features that are processed by the compiler. This means that any "dead", unreachable code is not represented in our database. While this may even be considered desirable for code that is unreachable because, e.g., the call to a method was commented out, this becomes a systematic problem for complex projects where code may only be compiled depending on options set at compile time. This is quite common amongst projects that use the GNU toolchain of configure and make. While that particular case does not concern us here, it remains possible that projects check for conditions in the compile environment that are not fulfilled in our setup, which may lead to certain files not being compiled.

For performance reasons, we limit the available processing time for each project's build to 25 minutes, as mentioned above. This means that we systematically lose all large projects that need longer to build. However, while the largest project in the corpus could not be built successfully due to internal errors, the second-largest project's build (having 12 701 537 lines of code) succeeded. Hence, we do not expect this artificial restriction to cause many problems. If an attempt to build a project fails, it is most often because of missing packages or coding errors.

Another problem is our lack of support for API versioning. As we downloaded the APIs that are made available on the build path during the course of our work, we generally provide projects with the most current API versions. This can cause problems for older projects that are no longer actively maintained and expect an older, incompatible API version. We currently do not have a reliable method of measuring the impact of this problem.

2.6.4 In the Lexeme-based Analysis

Compared to the analysis using the AST-based fact extractor, scanning the source code will generally show many more packages as being in use. There are two main uncertainties which skew our measurement:

- We do not check whether any features from an imported package are actually used.
- For on-demand imports, we do not analyse which features from within the imported scope are used.

Additionally, as mentioned in subsection 2.5.1.1, features that are referenced by their fully qualified names will not be detected by the fact extractor.

3 Tools

This chapter discusses implementation details of the software tools that were used for the analyses. Apart from the compiler plugin that was created for the AST-based fact extraction (which is described in section 3.1) and the scripts that were used for the lexeme-based approach (see 3.2), we needed to create an analysis environment and several auxiliary tools. The database that serves as the central storage meachnism in the analysis environment is described in section 3.3, and the rest of the chapter presents some auxiliary tools that were used in data preparation and testing.

3.1 The Compiler Plugin for AST-Based Fact Extraction

The fact extraction from the compiled projects is done with the help of a plugin that augments javac. The plugin needs to log detailed information about every method call in the compiled projects to the database. In order to do so, it needs to

1. be able to access information about the properties of a feature reference,
2. be invoked for every feature reference in a project and
3. be invoked automatically for every project in our corpus.

We combine three lesser known Java techniques to achieve this.

The analysis methods proper are provided by Java's Compiler API and the Compiler Tree API [java]. These APIs provide access to the abstract syntax tree (AST) that is created internally by javac during compilation. The AST holds a hierarchical representation of all feature references that the compiler encountered in a code unit. We use this data structure as the source of all information about feature references, especially type and scope information.

The Pluggable Annotation Processing API descibed in [Dar] augments javac with a plugin mechanism. We use this mechanism to give the plugin access to the compiler context. The compiler plugin extends javax.annotation.processing.AbstractProcessor. By setting SupportedAnnotationTypes to "*", indicating that the processor should react to all events in the code, we ensure that every feature reference is processed.

Finally, we needed a way to automatically call the plugin for every new project that was compiled by the build script. As we restricted ourselves to building ANT-based projects, we made use of ANT's ability to automatically load all JAR archives in ~/.ant/lib/. Annotation Processors are automatically detected if the containing JAR archive is supplied with appropriate service entries. In our case, this is done by the ANT buildscript of the fact extraction plugin, as shown in the following excerpt:

Listing 3.1: Setting service type information

```
1 <target name="jar" depends="compile">
2     <mkdir dir="build/jar"/>
3     <jar destfile="build/jar/FactExtractor.jar" index="true"
4             basedir="build/classes">
5         <service type="javax.annotation.processing.Processor"
```

```
6             provider="de.uniko.inf.api.Analyzer"/>
7         <indexjars/>
8     </jar>
9 </target>
```

Line 5 gives the service entry, while line 6 points to the main class of the plugin.

By placing the JAR file containing the fact extractor in ANT's `lib` directory and preparing a service description as mentioned, we can make sure that the plugin is started whenever ANT is invoked.

The following listing shows an excerpt from the plugin's main class. Apart from the parts of Java's APIs used in creating the analyzer, it shows the techniques involved in setting up and configuring a compiler plugin.

Listing 3.2: Creation of the plugin and injection into the compiler environment

```
1  import java.io.File;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.FileOutputStream;
5  import java.io.IOException;
6  import java.sql.Connection;
7  import java.sql.DriverManager;
8  import java.sql.ResultSet;
9  import java.sql.Statement;
10 import java.util.HashMap;
11 import java.util.LinkedList;
12 import java.util.Properties;
13 import java.util.Set;
14
15 import javax.annotation.processing.AbstractProcessor;
16 import javax.annotation.processing.ProcessingEnvironment;
17 import javax.annotation.processing.RoundEnvironment;
18 import javax.annotation.processing.SupportedAnnotationTypes;
19 import javax.annotation.processing.SupportedSourceVersion;
20 import javax.lang.model.SourceVersion;
21 import javax.lang.model.element.TypeElement;
22 import javax.lang.model.type.TypeKind;
23 import javax.tools.Diagnostic;
24
25 import com.sun.source.tree.ClassTree;
26 import com.sun.source.tree.ExpressionTree;
27 import com.sun.source.tree.IdentifierTree;
28 import com.sun.source.tree.MemberSelectTree;
29 import com.sun.source.tree.MethodInvocationTree;
30 import com.sun.source.tree.MethodTree;
31 import com.sun.source.tree.NewClassTree;
32 import com.sun.source.tree.Tree;
33 import com.sun.source.util.TaskEvent;
34 import com.sun.source.util.TaskListener;
35 import com.sun.source.util.TreeScanner;
36 import com.sun.tools.javac.code.Symbol;
37 import com.sun.tools.javac.code.Type;
38 import com.sun.tools.javac.processing.JavacProcessingEnvironment;
```

```
39 import com.sun.tools.javac.tree.JCTree;
40 import com.sun.tools.javac.tree.TreeInfo;
41
42 /**
43  * A plugin (an annotation processor) that is invoked for all
44  * source elements. This universal quantification is expressed
45  * by "*" in the annotation below.
46  */
47 @SupportedAnnotationTypes("*")
48 @SupportedSourceVersion(SourceVersion.RELEASE_6)
49 public class Analyzer extends AbstractProcessor {
50
51     /**
52      * Initialize the processor
53      */
54     @Override
55     public synchronized void init(ProcessingEnvironment procEnv) {
56         super.init(procEnv);
57         // Inject our TaskListener into the compiler context
58         t = new TaskListen();
59         ((JavacProcessingEnvironment)procEnv).getContext().put(
60                 taskListener.class, t);
61     }
62
63     /**
64      * This method needs to be overridden, but we do our
65      * analysis elsewhere. Returning false enables other
66      * annotation processors to again process the same
67      * annotations we encounter here, i.e. we don't claim them.
68      */
69     @Override
70     public boolean process(Set<? extends TypeElement> annotations,
71             RoundEnvironment r) {
72         return false;
73     }
74 //...
```

Lines 15 to 40 show the classes we use from the Pluggable Annotation Processing API and the Java Compiler and Compiler Tree APIs. The compiler plugin extends AbstractProcessor (l. 49), declaring its support for all annotation types (l. 47). The crucial step for accessing all information present in the AST is done in the init() method: The TaskListener t, which was created in line 58, is injected into the compiler context.

Afterwards, one would normally use the process() method for the actual work the annotation processor was intended to do. However, we use the TaskListener for that, so process() always returns just *false*. The reason is that, as specified in [Abs], annotation processors that return false here leave any annotations they come across for others to process. Hence, by returning *false* and not claiming any annotations ourselves, we enable the analyzer to co-exist with other annotation processors that may be integrated in the analysis environment in the future.

3.1.1 Tests

In order to check the accuracy of the AST-based analysis, a small program was written that instantiates Java classes in various ways. The test code, shown in listing 3.3 on page 18, covers

the relevant parts of the Java Language Specification (JLS). An overview over the possible ways to create new class instances is given in §12.5 of the JLS. From a programmer's viewpoint, there are at least five common ways to create objects in Java, all of which are implemented in the test program:

1. Class instantiation using the new keyword.
2. Class instantiation by using Reflection's Class.forName() mechanism
3. Cloning an existing object
4. Loading a class by explicitely using a ClassLoader and afterwards instantiating the class
5. Deserialization of a previously serialized object

In the test program, these instantiation methods are called on different classes. This facilitates error tracking. Apart from creating new objects, the program calls constructors and methods with different kinds of parameters to check whether parameter passing is detected correctly by the analyser. Finally, in order to check whether classes are identified correctly, some objects are created from classes that are located in subpackages and in an external JAR archive.

By wrapping this piece of sample code in a small application and analysing it with the compiler plugin, we could verify that it correctly extracts method calls in classes generated by every method allowed in the Java language, and that those classes are represented correctly in the ClassTree table of the database.

3.2 Scripts for the Lexeme-based Analyses

Because, in this project, the lexeme-based analysis was only intended as a supplement to the AST-based approach, we only needed to gather a very limited subset of the information available from the source files as described above in section 2.5.

We used a simple Java program to recurse through all 2121688 source files. For each file encountered, it would open the file, extract the import statements by regular expression matching and store them in the database's Imports table using JDBC for the database connectivity. Additionally, the program counts both the number of lines in each file and the number of non-comment lines and stores the values in the Files table.

3.3 The Database

All data gathered from the various sources – the SourceForge analysis scripts, the compiler plugin, the source code scanner, manual API tagging and the helper scripts – was stored in a relational database to make it accessible for further processing and analysis. The database modeling was rather straightforward initially: Each component of an object-oriented program – the program itself, packages, classes and methods – was modeled as a table, referencing the others in order to form a hierarchy. This data model is shown in Fig. 3.1.

From the point of view of our analysis, it does not make much sense to talk about programs as almost all usable information is derived from successful builds using the instrumented compiler. Consequently, the list of successfully compiled programs is stored in the *Build* table. Each build consists of *Scopes* that are visible to the compiler plugin through AST analysis. Scopes can either correspond to Java *Packages*, *Classes* or *Methods*.

The implementation of the database schema is somewhat different from this draft. An EER diagram of the database implementation is provided in figures 3.2 and 3.3, which had to be split

Listing 3.3: Implementations for the five ways to create an object in Java

```

/*
 * Five ways to initialize a class
 */

// canonical object generation
alpha = new Alpha();

// using Class.forName
try {
        Class<?> c = Class.forName
            ("de.unikoblenz.apianalyse.testsuite.Beta");
        beta = (Beta) c.newInstance();
} catch (Exception e) {
}

// cloning an existing object
gamma = new Gamma();
try {
        gammaClone = (Gamma) gamma.clone();
} catch (CloneNotSupportedException e) {
}

// using a ClassLoader
try {
        Class<?> d = this.getClass().getClassLoader().loadClass
            ("de.unikoblenz.apianalyse.testsuite.Delta");
        delta = (Delta) d.newInstance();
} catch (Exception e) {
}

// deserialization
epsilon = new Epsilon();
String filename = "epsilon.ser";
try {
        FileOutputStream fos = new FileOutputStream(filename);
        ObjectOutputStream out = new ObjectOutputStream(fos);
        out.writeObject(epsilon);
        out.close();
} catch (Exception e) {
}
try {
        FileInputStream fis = new FileInputStream(filename);
        ObjectInputStream in = new ObjectInputStream(fis);
        epsilonAgain = (Epsilon) in.readObject();
        in.close();
} catch (Exception e) {
}
File f = new File(filename);
f.delete();
```

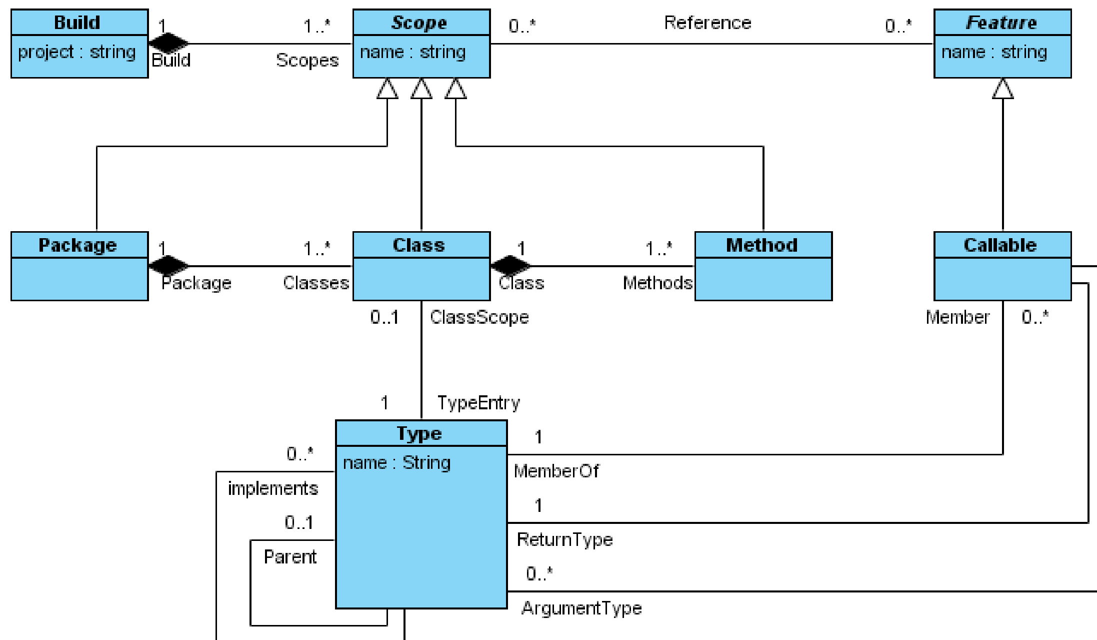


Figure 3.1: The data model behind the internal representation of programs

because of readability concerns. The most obvious difference, compared with the design draft, is the larger number of tables used in the implementation. The main reason for this is that several helper tables were necessary to store metadata. Additionally, for performance reasons, parts of the database were denormalized later. While this increased the size of the database, query performance improved markedly.

3.3.1 Contents of the tables

This section gives an overview of the contents of the tables in the database and some of their relations. Tables are mentioned in the order in which data acquisition and processing was done.

3.3.1.1 Data obtained from the compiler plugin

Feature The *Feature* table lists all features encountered by the instrumented compiler, gives their names and modifiers along with the ID of the class the feature was found in. The *featuretype* field corresponds to the *Callable* class in the data model, and currently only differentiates between methods and constructors. If the feature was determined to be part of a known API, *partOfAPI* stores the ID of that API, referencing the table *KnownAPIs*. *partOfAPI* is "0" when a Feature is not part of a known API and "null" if it has not yet been checked. This is also how *partOfAPI* is used in the other tables containing the field.

Scope A table containing all the scopes found in the source code. This table is linked to *Feature* via *ScopeFeature*.

ClassTree Contains all classes and interfaces (refered to via *ClassTree_Interfaces*) encountered by the instrumented compiler. Each class or interface can be a part of a known API, which is again stored in a field called *partOfAPI*. The class hierarchy is represented by

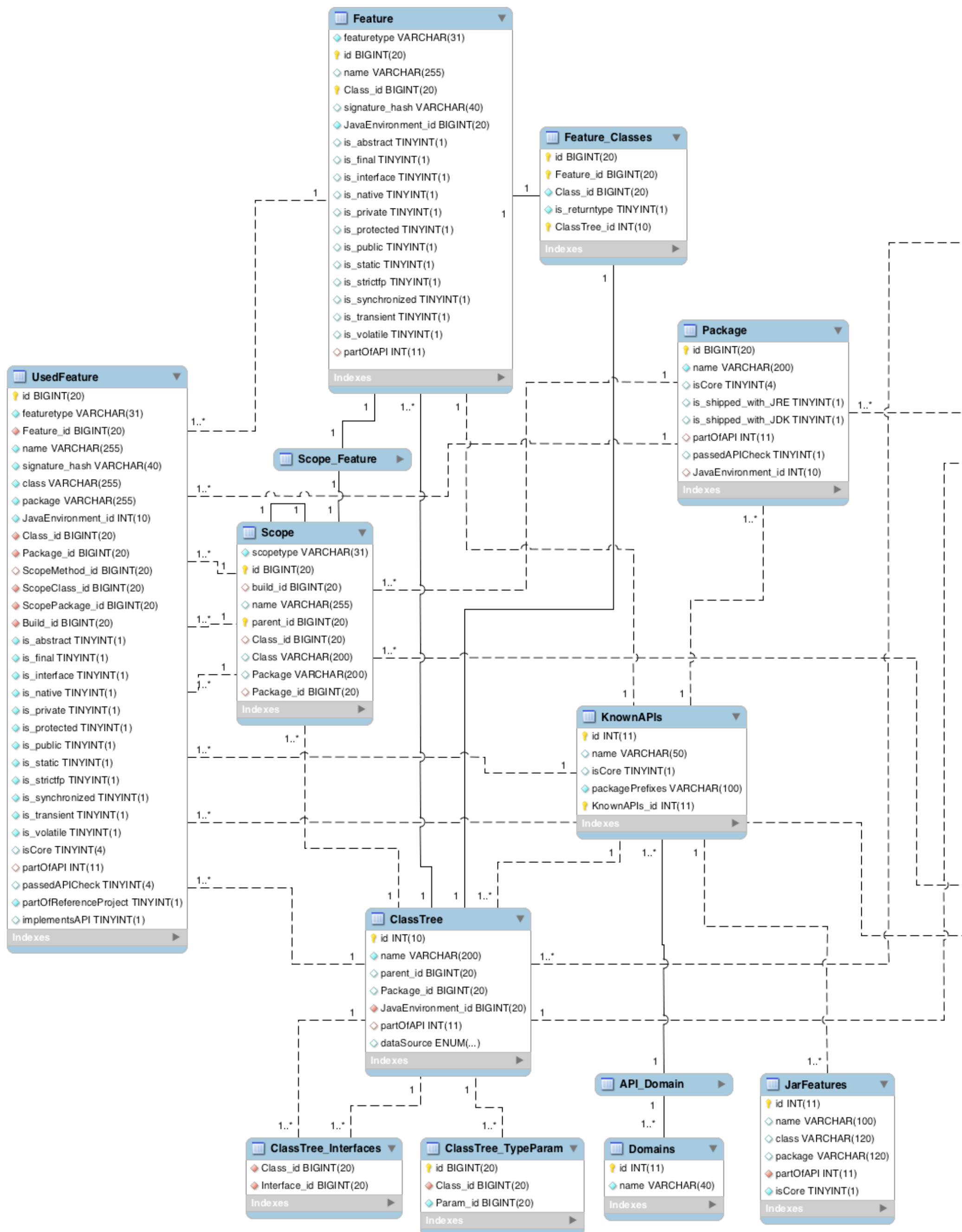
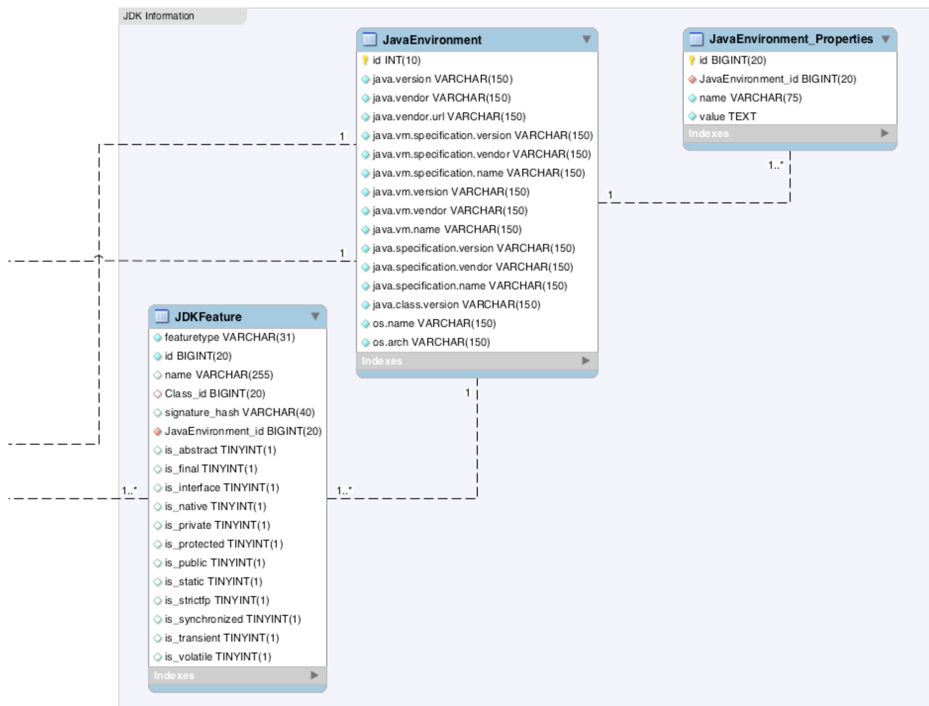


Figure 3.2: ER-diagram of the main database (continued in fig. 3.3). This part shows tables concerned with code representation.



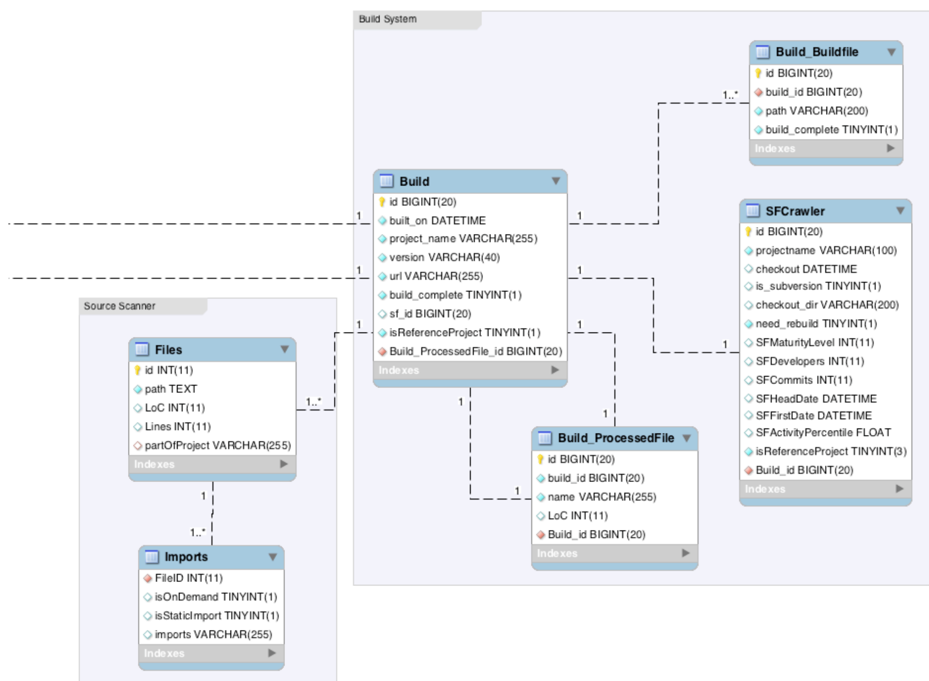


Figure 3.3: ER-diagram, continued. This part shows tables for metadata storage.

storing a link to a classes' parent in the *parent_id* field. This table is linked to *Feature* via *Feature_Classes*.

Package Stores all packages encountered by the instrumented compiler. For each package, some meta information about its origin is stored in the two boolean fields *is_shipped_with_JRE* and *is_shipped_with_JDK* as well as the *JavaEnvironment_id* field which references the *JavaEnvironment* table. This field and the corresponding table are intended for future extension to different JDKs. The boolean *isCore* field is true if the package is contained in the Core Java distribution. *passedAPICheck* denotes whether a package passed a scripted test for API-like usage patterns, indicating that this package is likely part of an API that we did not manually include in our set of known APIs.

UsedFeature In order to be able to write more concise queries, the most important fields of the aforementioned tables were collected in this table. For example, there is no need to create a join across the Feature, ClassTree and Package tables just in order to use the fully qualified name of a feature. This, however, also means the database is no longer in normal form. Despite this, a set of scripts ensures that data integrity is not violated.

3.3.1.2 Data obtained from analyses of binary distributions

KnownAPIs Lists the APIs that were analyzed with the method described in sections 2.3.4 and 3.4, giving their name and characteristic package prefixes as well as another *isCore* tag. Additionally, during the manual analysis, each API is assigned to a domain:

Domain, API_Domain An API belongs to an application domain such as *XML processing, graphics, networking* etc. The set of domains was created ad-hoc and are free-form (compare section 2.2.1). Each entry in *KnownAPIs* is linked, via *API_Domain*, to an entry in *Domain*.

JarFeatures Lists all features found in the binary distributions of APIs (i.e., the jar files the authors provide for download), regardless of whether they are used in project code or not. Unlike *Feature*, this table stores methods with different signatures only once.

JDKFeature Lists all features found in the JDK as shipped by SUN (we used version 1.6.10).

3.3.1.3 Data obtained from the text-based scanner and the build system

Build Contains SourceForge-related meta information about each project we built. The most important field is *referenceProject*, which indicates whether the project fulfils the criteria for reference projects as defined in section 3.6.

SFCrawler, Build_ProcessedFile These are helper tables that store internal information for use by the downloading scripts and the build environment.

Files For each source file from the projects, this table lists the filename, total lines and lines of code (which are defined as the lines that are neither comments nor whitespace).

Imports The lexeme-based scanner stores the results of the import detection process (as described in subsection 2.5.1.1) in this table.

3.4 Manual API Tagging

In order to provide more detailed and accurate information than would have been available by automated methods, a selection of widely-used APIs was manually prepared for analysis.

We decided whether or not to analyze and tag an API based on its usage: First, we created a list of all packages that were not yet tagged, ordered by usage frequency. Then, beginning with the most often used packages, we categorized them until more than half of all packages that passed the automatic API candidate test were categorized (compare table 4.3 on page 35). Additionally, all APIs from the Apache Commons repository available as jars were tagged because they are used very widely (i.e. by a lot of projects, not necessarily with a lot of feature references). In the end, 77 APIs were downloaded (in binary distribution form).

Each of those was then processed with the help of a script that scanned the contents of JAR files using Java's Reflection API and entered every method and the class structure in the table JarFeatures in the database. In addition, for each Feature stored, the script added information about which API it belonged to, and asked the user to assign a domain label to the API (for a definition of the term "domain" in this context, see section 2.2.1 on page 6).

The result of this manual categorization is a list of APIs, a mapping between APIs and domains and a mapping between APIs and substrings from package names. Those packages that have a name which starts with one of those known substrings are marked as belonging to the corresponding API. Inside the database, a dedicated table KnownAPIs contains the APIs' names and the field partOfAPI in the Package table stores a reference to an API from KnownAPIs if the package in question has been identified to be part of a known API. The mapping itself, and hence the result of the manual examination of the APIs, is stored in a plain SQL file that is generated by the script. After the tagging is done, this file is manually imported into the database.

This procedure marks packages based on substring matching on the package names, so there may be errors in its results if programmers violate SUN's Java coding guidelines and use namespaces that are not unique. SUN recommends using one's internet TLD and domain name as a namespace, so this problem may occur if a programmer either uses another namespace basis or a domain gets used by different people for different projects over time.

As an example, let us assume that both an API and a project use a package called logging. Although both may be completely independent of each other, once we see the logging package in the API, the logging package of the application would be matched, too, because simple substring matching is unable to differentiate between the two.

The APIs that were manually examined up to now adhere to the naming conventions; the mentioned problems are, thus, of no concern here. The only exception is JUnit in version 3, which uses junit.* as a package namespace. This namespace, however, is not used by any other projects in our database.

Because we discovered that many APIs include verbatim copies of parts of other APIs or non-API code, mostly in JARs with sample code, the script offered the user an option to exclude certain packages from analysis. This reduced the number of features from sample code or incorrectly tagged features in the database.

After processing the contents of the JAR files, the entire binary distribution of the analysed API was copied to a central directory where it would later be made available to the build script. Each project that we compiled was thus able to access any of the analysed APIs, and every call to a feature from one of those APIs could be logged correctly. As mentioned in section 2.4.1, this allowed around 200 additional projects to be built and analyzed successfully.

To give an idea of the results of this process, table 3.1 on page 25 shows the list of all APIs we processed in this manner, along with some basic information about them. The Core column indicates whether an API is a part of Core Java (see 2.3.5.1 on page 9). The feature, package

Algorithm 3.1 Detection of packages with API-like usage patterns

```
for each Package p in the database:

    p.passedAPICheck = true
    if p is the default package:
       p.passedAPICheck = false
    if there is no class with at least one public method in p:
       p.passedAPICheck = false
    if there is only one project referring to features from p:
       p.passedAPICheck = false
    if there is at least one project referring to features from p
      and p is not compiled in that project:
       p.passedAPICheck = true
```

and project counts refer to the number of features that were seen by the compiler plugin, thus reflecting the amount of use that the respective API sees in our corpus. Note: these numbers are independent of the number of features an API has in total.

The contents are ordered by the number of feature references in order to reflect the initial selection mechanism.

3.5 Automatic API Detection

The large number of packages in the corpus (46145) makes it basically impossible to manually classify them all into API, project and Core Java packages. Hence, we developed an algorithm (given in algorithm 3.1) for detecting packages that are used like a library package would.

This algorithm correctly marks most packages that are used as if they were included in an API. It most notably fails in the case that one of the projects we examine is itself an API or a framework. If no other project uses features from a package of this project, said package will fail the test although it should actually pass.

Consider, for example, an API A containing a package org.example.A.foo. If we have the source code of A among the projects we examined, but no feature from org.example.A.foo is used in any of our projects, org.example.A.foo will fail the test by the third if-clause of the algorithm, and as there are no other references to it, the fourth will not rectify this mistake. An error like this, however, will only affect one package at a time, so both org.example.A itself and org.example.A.bar will be examined independently and will pass the test, if there are external references to their features.

3.6 Reference Projects

As with other selection and tagging processes in our corpus, we aimed to automate the process of finding reference projects as much as possible.

API	Domain	Core	Features	Packages	Projects
Swing	GUI	yes	20432	96	716
Java Collections	Collections	yes	18584	0	1374
OFBiz	e-Business	no	10674	415	3
AWT	GUI	yes	10008	36	754
SWT	GUI	no	5693	34	30
Express4J	GUI	no	5133	128	2
Hibernate	Database	no	3495	81	63
JUnit	Testing	no	3487	48	233
Lucene	Search	no	2132	80	36
MySQL Connector/J	Database	no	1830	18	8
Commons Collections	Collections	no	1588	63	37
Core XML	XML	yes	1284	42	413
j2ssh	Networking	no	1272	34	4
GWT	Web Apps	no	1156	31	13
Jena	Semantic Web	no	1063	51	20
Commons Lang	Other	no	1051	12	93
Reflection	Other	yes	968	4	560
log4j	Logging	no	891	12	254
Bouncy Castle Crypto	Security	no	884	28	16
Commons Net	Networking	no	786	16	10
JDOM	XML	no	712	18	86
JMF	Media	no	696	25	28
JFace	GUI	no	647	16	10
Commons Math	Other	no	619	19	6
Commons Logging	Logging	no	511	4	151
SAX	XML	no	484	4	310
LWJGL	GUI	no	415	23	10
Struts	Web Apps	no	300	16	26
Axis	Webservices	no	299	34	30
DOM	XML	yes	280	13	324
XMLBeans	XML	no	271	7	9
Xerces	XML	no	263	13	42
TestNG	Testing	no	253	3	14
Axis2	Webservices	no	243	28	5
dom4j	XML	no	218	7	37
JNDI	Networking	yes	211	6	101
Commons CLI	Other	no	184	7	32
WSMO4J	Webservices	no	182	21	2
Commons Beanutils	Other	no	176	2	51
Commons Digester	XML	no	161	4	20
RMI	Networking	yes	158	6	64
XOM	XML	no	145	4	5
Berkeley DB	Database	no	143	8	9
Commons Codec	Other	no	140	6	27
jMock2	Testing	no	132	10	15
Java 3D	GUI	no	125	11	12
Commons IO	IO	no	117	3	34
AXIOM	XML	no	89	10	4
Batik	GUI	no	83	17	10
Commons Configuration	Other	no	72	3	9
JAI	GUI	no	65	5	9
Commons FileUpload	Networking	no	64	5	31
Commons Pool	Other	no	55	4	14
Commons DBCP	Database	no	55	4	15
jogl	GUI	no	53	4	9
GNU Trove	Collections	no	47	2	4
Commons DbUtils	Database	no	36	2	5
QuickFIX	e-Business	no	36	7	1
Commons Email	Networking	no	30	1	5
Commons Betwixt	Other	no	30	4	3
XMLPull	XML	no	24	2	7
Jaxen	XML	no	24	5	6
Guice	Other	no	12	5	2
JavaHelp	GUI	no	11	5	8
StAX	XML	no	9	2	2
Struts2	Web Apps	no	8	5	2
Commons Chain	Other	no	7	2	1
Java Expression Language	Other	no	6	1	2
Commons Transaction	Other	no	4	2	1
Commons Discovery	Other	no	3	1	2
Xalan	XML	no	3	3	3
Commons Proxy	Other	no	0	0	0
Commons Primitives	Other	no	0	0	0
Commons Exec	Other	no	0	0	0
Commons EL	Other	no	0	0	0
Commons Daemon	Other	no	0	0	0
Commons Attributes	Other	no	0	0	0

Table 3.1: For each manually tagged API, this table shows its name and domain, whether it's a part of Core Java, the number of feature and package references that were seen in active use in the corpus and the number of referencing projects.

3.6.1 Data Sources

When we extracted the data, SourceForge provided (amongst other data that was not used for this task) the following statistical information for each project:

Activity percentile This value is a rough approximation of the developers' activity in a project. It is given as the percentage of projects registered on SourceForge that have seen less activity in the past week than the current one (i.e. the most active project in a week has an activity percentile of 100%). SourceForge itself does not explicitly state what interactions with the site constitute an activity that is counted into this value.

Development status Developers can rank the perceived maturity of their projects themselves using one of the following categories:

1. Planning
2. Pre-Alpha
3. Alpha
4. Beta
5. Production/Stable
6. Mature
7. Inactive

Number of developers The number of registered SourceForge members that contribute to a given project. This value varies with time, and represents only the number of registered developers at the time of viewing.

Number of commits The number of commits to a project's Subversion repository. This number may be skewed if a project changed from another revision control system and/or another hosting provider to SourceForge's Subversion service during development.

Dates of first and most recent commits The Subversion repository stores timestamps for every commit. In order to be able to calculate the time for which a repository was in active use, we extracted the time and date of the oldest and most recent commit from the repository web pages.

The data was retrieved by HTML-scraping the project pages. During the site's redesign in late June 2009, however, many of these values were removed from the project pages. They are still accessible from the "Trove" software map, the site's main overview feature. Hence, the approach we took is no longer feasible and will need to be modified slightly to accomodate for the new data sources.

Alternatively, database dumps containing this information can be obtained from the FLOSSmole project ([HCC06], [Flo]), which, however, was unknown to us when we started the data collection process.

3.6.2 Selection Criteria

Based on the aforementioned data, we require a reference project to fulfill the following criteria:

- The project rates itself "stable" or "mature" in SourceForge's self-rating system
- The Subversion repository has been in use for more than two years
- There have been more than 100 commits to the source repository

While these criteria are somewhat arbitrary, they create a rather homogenuous group of 60 projects that fit the initial informal criterion of well thought-out, actively developed and usable software. For this classification, it was more important to create a control group that does not include any obviously incomplete or immature projects, so our classification approach should be considered as incomplete: It is rather successful at avoiding false positives (i.e. immature projects that are added to the group of reference projects), but is likely to produce many false negatives (i.e. mature projects that are not detected).

The other metadata mentioned above were collected in order to have alternative maturity indicators available. We did however notice that some are not helpful for our classification needs. The activity percentile was left out because it was unclear which events on SourceForge are counted into that value, and mostly because it seemed to include user activity on the project page. That, however, can be triggered by many things that are independent of development itself. The number of developers is, in itself, also not an indicator of a well-maintained project, as shown in [Kri02].

3.6.3 Script-based Tagging

The base data listed above were extracted from SourceForge's project sites with a Python script. The script initially generates a list of the names of all projects that we downloaded using Subversion. For each project, we fetch the HTML source of its project page (`http://sourceforge.net/projects/[PROJECTNAME]`) and process it using the BeautifulSoup library [Bea]. The values for activity percentile, development status, number of developers and commits and the dates for the oldest and most recent commits are entered in the database, in the SFCrawler table (compare the ER-diagram, fig. 3.2 on page 20).

A small SQL script reads those values and sets the isReferenceProject flag for those projects that meet the criteria.

3.7 Clone Detection

From the perspective of the feature detection method we use, there are two different cases of code duplication that may lead to errors in the analyses.

Most previous work on code clone detection (as discussed, among others, in [RCK09]) focuses on the detection of code duplicates in single programs or the detection of code sections with similar functionality. We face different situations and, hence, can not rely on most existing clone detection approaches.

3.7.1 Inclusion of API Code in other APIs

There are several cases where binary API distributions include parts of other APIs. For example, the XOM distribution, xom-1.2.1.jar, contains a set of jaxen classes (see fig. 3.4). Similarly, the Apache Foundation's Axis2 library is packaged with several other APIs, which are shipped using a naming convention like axis2-json-1.4.1.jar.

While this is not a problem for statistical analyses per se, one must be careful to correctly seperate those APIs from each other. We opted for a pragmatic solution, based on manual exclusion of parts of JAR archives, discussed in detail in section 3.4. Through this process, we avoid counting features for the wrong APIs, which would skew the relation of feature reference counts between APIs, as well as tagging features as belonging to a wrong API, which would give incorrect API size measurements.

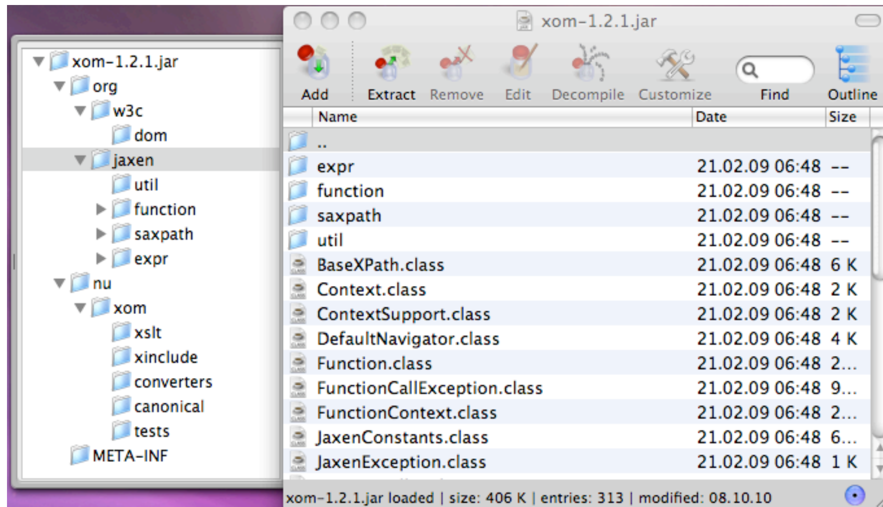


Figure 3.4: A JAR viewer showing jaxen classes in xom-1.2.1.jar

3.7.2 Inclusion of API Code in Project Code

Instead of using the jar files that most APIs are distributed as, many client programmers include API source code directly into their programs. The resulting mix of client and third-party-code in projects makes it difficult to correctly assess the status of a given piece of code in our corpus.

Listing 3.4 shows an example of such a situation. The tsg project is one of several that copy API code into their own codebase, in this case 21 classes from GNU Classpath's Collections API. The listing shows the output of an sdiff run comparing the original version from the Classpath project, used to create the binary API distribution in our corpus, and the version included in the tsg source tree.

A look at the date differences in the file header suggests that the project authors used an older version of the file than included in the 0.9.8 release of Classpath that was current at the time the diff was taken. Apart from the FSF moving house, we notice that the project authors applied small changes to the class, removing four import declarations from the file, modifying the type of the loadFactor and load parameters, removing generics and commenting out two statements. The meaning of their comment line "NOT IN MIDP" is unclear.

It is not certain whether the file that was originally copied already had the type parameters included, so whether these differences are due to an intentional simplification of the file is not clear. However, even just changing the parameter types is an alteration that makes the local API version incompatible with the original one.

While the reason why programmers do not rely on jar archives for library distribution is unclear, this phenomenon should not be ignored. With API features copied in source form into project code, feature reference counts of affected libraries will appeat to be too low. This is because our analysis scripts rely on the tagging of API features as such, and because that tagging works on the level of downloaded binary API distributions, source copies won't be properly tagged. In order to improve those statistical analyses that rely on the distinction between project and API code, it would thus be necessary to detect cases where API features occur in projects as a result of such copying.

Because of time constraints, however, we do not yet apply appropriate detection techniques.

Listing 3.4: This is the sdiff output comparing two implementations of the HashSet class. On the left is the API version as delivered with GNU Classpath 0.9.8, on the right the same file as included in the "tsg" project. Only lines that differ between the two versions are shown.

```
1
2
3      Copyright (C) 1998, 1999, 2001, 2004, 2005  Free Sof|    Copyright (C) 1998, 1999, 2001, 2004 Free Software
4    Free Software Foundation, Inc., 51 Franklin Street, Fif| Free Software Foundation, Inc., 59 Temple Place,
5    02110-1301 USA.                                        | 02111-1307 USA.
6    package java.util;                                     | package org.placelab.collections;
7    import java.io.IOException;                            | //import java.io.IOException;
8    import java.io.ObjectInputStream;                      | //import java.io.ObjectInputStream;
9    import java.io.ObjectOutputStream;                     | //import java.io.ObjectOutputStream;
10   import java.io.Serializable;                           | //import java.io.Serializable;
11    * @author Jon Zeppieri                                |  *
12    * @author Eric Blake (ebb9@email.byu.edu)             |  *
13    * @see TreeSet                                        <
14    * @see Collections#synchronizedSet(Set)               <
15    * @see LinkedHashSet                                  <
16    * @status updated to 1.4                              <
17   public class HashSet<T> extends AbstractSet<T>         | public class HashSet extends AbstractSet
18     implements Set<T>, Cloneable, Serializable           |   implements Set, Cloneable//, Serializable
19     private transient HashMap<T, String> map;            |   private transient HashMap map;
20     public HashSet(int initialCapacity, float loadFactor)|   public HashSet(int initialCapacity, int loadFactor)
21     public HashSet(Collection<? extends T> c)            |   public HashSet(Collection c)
22     public boolean add(T o)                              |   public boolean add(Object o)
23     public Object clone()                                |   public Object pclone()
24       HashSet<T> copy = null;                            |     HashSet copy = null;
25           copy = (HashSet<T>) super.clone();             |         copy = (HashSet) super.pclone();
26       copy.map = (HashMap<T, String>) map.clone();       |     copy.map = (HashMap) map.pclone();
27     public Iterator<T> iterator()                        |   public Iterator iterator()
28     HashMap init(int capacity, float load)               |   HashMap init(int capacity, int load)
29                                                          >   /* NOT IN MIDP
30       Iterator<T> it = map.iterator(HashMap.KEYS);       |     Iterator it = map.iterator(HashMap.KEYS);
31                                                          | */
32                                                          >   /* NOT IN MIDP
33         map.put((T) s.readObject(), "");                 |       map.put(s.readObject(), "");
34                                                          >   */
```

3.8 Querying and Plotting

In order to facilitate updating result graphs and tables, we aimed to automate the analysis of the data in the database as far as possible. It was, hence, scripted.

The main analysis script was written in Python. For every graph or table desired, it includes a method that queries the database, performs any calculations necessary, outputs the result set to a .csv file on disk and prepares the output graph or table.

The graphs were mostly created using the open source graphing tool gnuplot[1], which we chose mainly for its scriptability.

For tables, we created the appropriate LaTeX code directly in the Python script.

3.8.1 Analysis Scripts: Sample Workflow

The analysis script provides some support structures for the analysis methods proper. Mainly, it connects to the central database and offers a cursor object that enables methods to send SQL queries to the database.

In order to speed up the execution of the script, each method is run in its own thread. Using one thread for each physical CPU core present, we were able to parallelize most time-consuming queries. Instead of calling the analysis methods directly, we use a start_thread method that associates it with the controlling semaphore, passes the name of the file or files it should work on and, on finishing, prints an appropriate log message.

Listing 3.5: The main elements of the analysis script. The analysis methods proper are omitted for brevity, as well as several internal helper methods.

```
1  import MySQLdb
2  import csv
3  import Gnuplot
4  import os
5  import subprocess
6  import threading
7
8  #...
9
10 def get_mysql_cursor():
11     db = MySQLdb.connect(user="....", db="api")
12     db.autocommit(True)
13     return db.cursor()
14
15 def start_thread(semaphore, thread_list, callable, *args, **kwargs):
16     def _():
17         with semaphore:
18             endmsg = kwargs.get("endmsg")
19             if endmsg:
20                 del kwargs["endmsg"]
21             callable(get_mysql_cursor(), *args, **kwargs)
22             if endmsg:
23                 print endmsg
24     t = threading.Thread(target=_)
25     t.start()
```

[1] http://www.gnuplot.info

```
    thread_list.append(t)
    return t

def main():
    sem = threading.Semaphore(4) #one thread per processor core
    threads = list()
    #...
    start_thread(sem, threads, APIFeaturesToProjectSize,
            "APIs/APIFeaturesToProjectSize",
            endmsg="API␣features␣vs␣project␣size␣done")
    #...
    print "All␣Threads␣started."
    for thread in threads:
        thread.join()
    print "Run␣complete."

if __name__ == '__main__':
    main()
```

Listing 3.6 shows one of the methods that generate the actual plots. It is used to create figure 4.6c, a plot showing the number of references to API features in relation to the projects' size, and gives a good impression of a typical simple query.

The method is called with two parameters: the shared cursor object for database access and the desired output filename. It uses the cursor to query the database, using embedded SQL code and, without further calculations, stores the results in a CSV file under the specified filename.

The final line of the Python script calls gnuplot in order to process the created CSV data file.

Listing 3.6: Python method

```
def APIFeaturesToProjectSize(cursor, filename):
    cursor.execute("""
        SELECT
            B.id,
            B.Size as 'Size',
            A.APIFeatures as 'APIFeatures',
            B.isReferenceProject
        FROM
            (SELECT b.id,
                count(distinct uf.Feature_id) AS 'APIFeatures'
             FROM Build b, UsedFeature uf
             WHERE uf.Build_id = b.id AND uf.partOfAPI != 0
             GROUP BY b.id) A
        LEFT JOIN
            ((SELECT b.id,
                count(*) as 'Size',
                count(distinct uf.Feature_id) AS 'AllFeatures',
                b.isReferenceProject
             FROM Build b, UsedFeature uf
             WHERE uf.Build_id = b.id
             GROUP BY b.id)) B
        USING (id);
        """)
```

```
24      data = list()
25      for bid, size, feat, ref in cursor:
26          data.append((bid, size, feat, ref))
27      writeToCsv(["bid size feat isReference".split()] + data, filename)
28      plot(filename)
```

The gnuplot plotfile for the example graph is shown in listing 3.7. Using standard gnuplot options, this file generates an x-y-plot from the intermediate CSV file. Splitting up the generation of the results file and the generation of the graph has the advantage that several plots can easily be generated from the same data file – for example, generating a version of this figure with a linearly scaled y-axis would only require unsetting the "log x" option, pointing gnuplot to a second output file and calling replot.

Listing 3.7: Gnuplot plotscript

```
1  # XY-Plot "APIFeaturesToProjectSize"
2  #
3  # This plot shows the number of different API features used in a project
4  # in relation to the project's size
5  #
6  # Assumed data file ordering:
7  # name Size APIFeatures isReferenceProject
8  # 1    2    3           4
9
10 reset
11
12 set terminal postscript eps color
13 set output "APIFeaturesToProjectSize.eps"
14 set datafile separator ","
15 set nokey
16 set log x
17 set xlabel "Project Size"
18 set ylabel "Number of API features used"
19 set yrange[0.0001:]
20
21 plot 'APIFeaturesToProjectSize.csv' using ($2):($4 == 0 ? $3 : -15) lc 3,\
22 '' using ($2):($4 == 1 ? $3 : -15) lc 1 pt 7
```

After the plot has been generated, the thread containing the analysis method terminates.

By scripting the analyses in this way, evaluating changes to the queries became relatively easy. However, due to gnuplot's limited abilities, some graphs still had to be generated manually using different tools.

4 Results

After the build process was done, the actual measurements could be taken. By that time, data from the AST-based analysis were available by querying the database. As the compiler plugin provided information about types and inheritance, as well as the source of every feature encountered, most of the information from the original project sources was available there. This facilitated some measurements like the coverage analysis, while the lexeme-based method provided a simple way to gather source file statistics.

The graphs and tables in this chapter are based, unless otherwise noted, on the AST-based analysis. We often use feature reference counts in these measurements; what we consider to be a feature reference is defined in subsection 2.2.2.

4.1 Measurements of Corpus Size

In order to give an impression of the size of the corpora we worked with, we give some basic figures for both the set of downloaded and built projects. An overview of the set of manually tagged APIs along with some metrics was given in table 3.1 on page 25.

The data in table 4.1 is based on the lexeme-based analysis (compare section 2.5) of all source files from all downloaded projects. These numbers give an impression of the size of the set of downloaded projects that we tried to build.

Metric	Value
Downloaded projects	6286
Source files	2121688
Lines of code	377640164
LoC (excluding comments)	264536500
Import statements	14335066

Table 4.1: Metrics of the source code corpus

Table 4.2 shows some numbers that describe the corpus of built projects. They were created by the AST-based analysis, which means that they encompass only code from compiled Java files that belong to projects which could be built successfully. Any Java files with unreachable code, sample code etc. that may have been part of the downloaded projects are not included in these counts.

The tables also demonstrate the different view that the AST-based analyzer has on the source code: The corpus size is not given in units related to files in a file system or lines of code, but rather in units like methods and feature references, which directly correspond to the representation of the code as an AST.

Metric	Value
Built projects	1476
Packages	46145
Classes	198948
Methods	1397099
Feature references	8163083

Table 4.2: Metrics from the AST-based analyzer for the corpus of built projects

4.2 Sources of Features in Project Code

In order to understand the relative importance of API features in the corpus, we split the set of all feature references that were recorded by the AST-based analysis into partitions according to the features' origin. The sources of the features are determined on the package level: After the AST-based analysis, the database contained, for each feature reference, the fully qualified name of the respective feature. Afterwards, we applied the tagging processes mentioned below to the database, tagging the features from the first three categories as such. Together with features that miss an explicit package declaration and feature that could not be attributed to one of the other categories, we distinguish five possible origins for a feature here:

Core Java Features in this partition are from the JDK we used in the analysis. For a definition of Core Java, see section 2.3.5.1.

Tagged APIs These features are from APIs that we tagged using the manual process described above in section 3.4. APIs consisting of Core Java Features, like Swing, AWT or Java Collections, count towards the Core Java partition in the following measurements.

Check passed This partition comprises features from APIs that were discovered by the automated process described in section 3.5.

Default Features from the default package

Others These are features from packages that fall into none of the other categories. Generally, these are project-internal packages.

Figure 4.1 shows the percentage of features from each of these sources in our corpus. The segments of the inner ring of the ring chart show the relative percentage of features from the five different code sources. In order to give a rough impression of how the usage of features differs between different packages, the six most often used packages from both Core Java and the manually tagged APIs are shown seperately in the diagram's outer ring. It is noteworthy that the java.lang package alone gets more feature references (1 200 672) than all manually and automatically detected API features combined (1 192 338). Similarly, there are more features from the default package than from the largest API package.

In order to get an impression of the "popularity" of APIs, two approaches suggest themselves. One may consider the number of feature references to an API, or the number of different projects that use it. These two measurements are given in Figure 4.2. The upper chart shows the number of feature references to the 25 most referenced APIs, in descending order. The bottom chart shows, using the same ordering, how many different projects use features from those APIs. By comparing the two charts, it becomes clear that the two metrics are independent of each other: Some APIs, like Express4J and OfBiz, concentrate very many references in only two or three

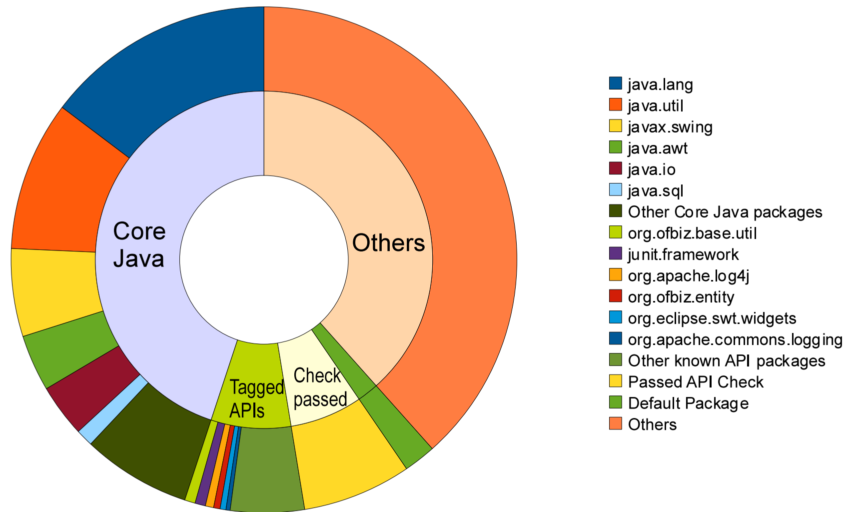


Figure 4.1: The distribution of features from Core Java, API and project code across the entire corpus

different projects, while others are spread out over more than 100 projects. For comparison, the same measurements, but for domains, are presented in Fig. 4.3a on page 37.

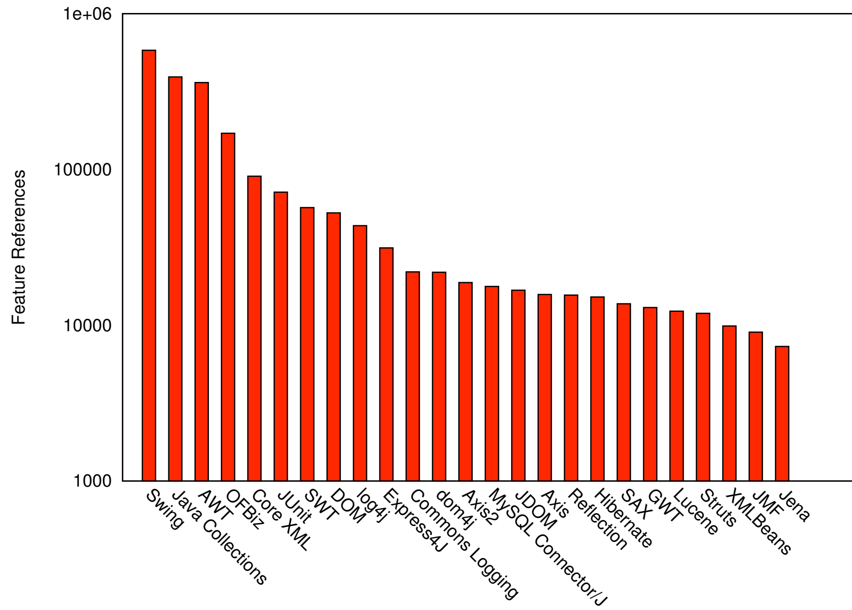
Table 4.3 shows the corresponding percentages for the different partitions. Perhaps surprisingly, we see that API code contributes only a total of 14.6% of all the code we see in the corpus. If, however, we restrict ourselves to the set of reference projects, which is intended to hold mature projects with reasonably good architecture (compare section 2.3.3), we note that the percentage of features from APIs rises slightly to 16.47%, while the percentage of Core Java features drops by 3%.

Source	*In All Projects*	*In Reference Projects*
Core Java	44.93%	41.16%
Manually tagged APIs	7.65%	11.83%
Automatically tagged APIs	6.94%	4.64%
Default package	2.06%	1.71%
Others	38.43%	40.65%

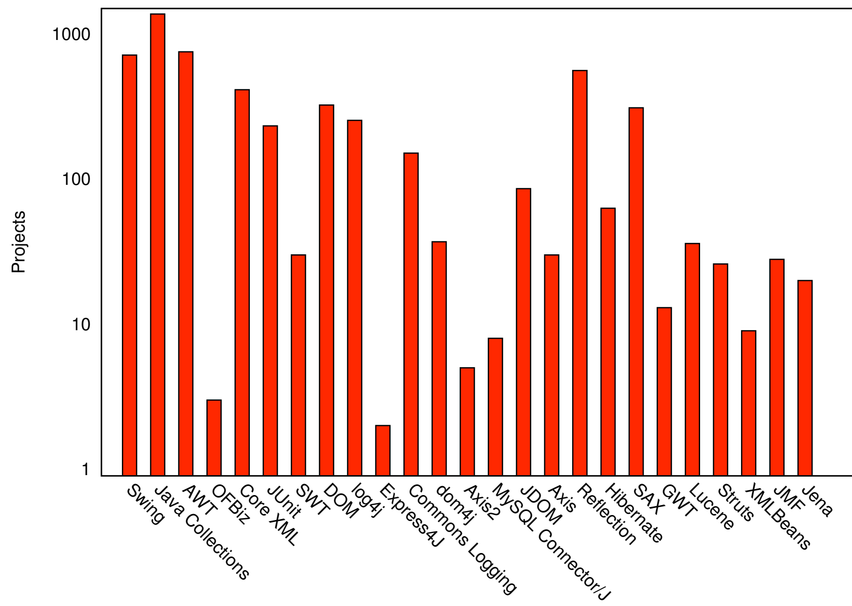
Table 4.3: Percentages for code sources in the corpus

While we observe that the percentage of API features in a project is higher for projects from our reference group, there is no such dependency between API usage and project size. While one might assume that larger projects use APIs more intensely, because they have more reuse opportunities, figure 4.4 shows, using the 15 largest projects in the corpus, that only a relatively small percentage of large projects' code are features from third-party APIs. Just like in the average case for the entire corpus, these large projects mostly use features from Core Java or project-specific code.

There are, however, other properties of the projects in our corpus that show clear relations to project size. In figure 4.5, we plot the number of packages the projects use in relation to their

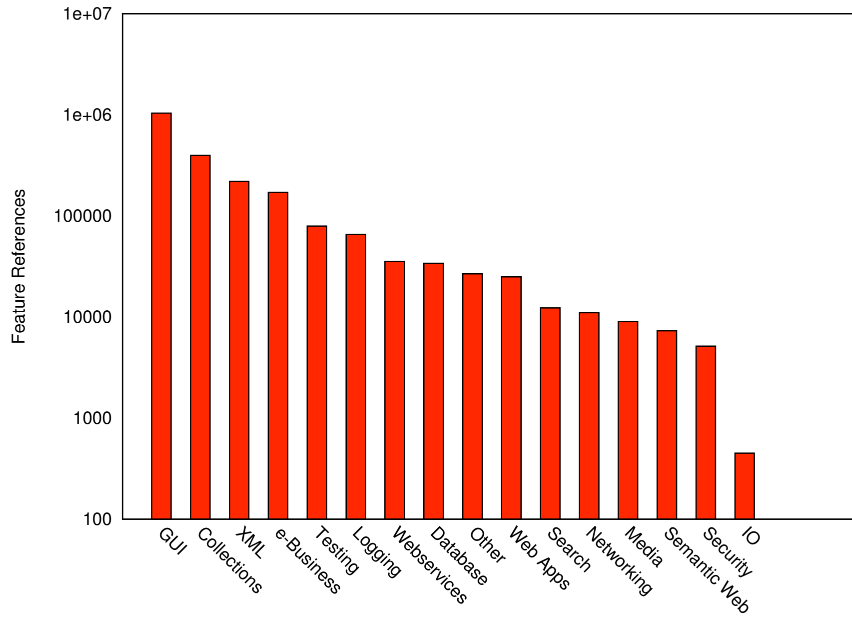


(a) Number of feature references to an API

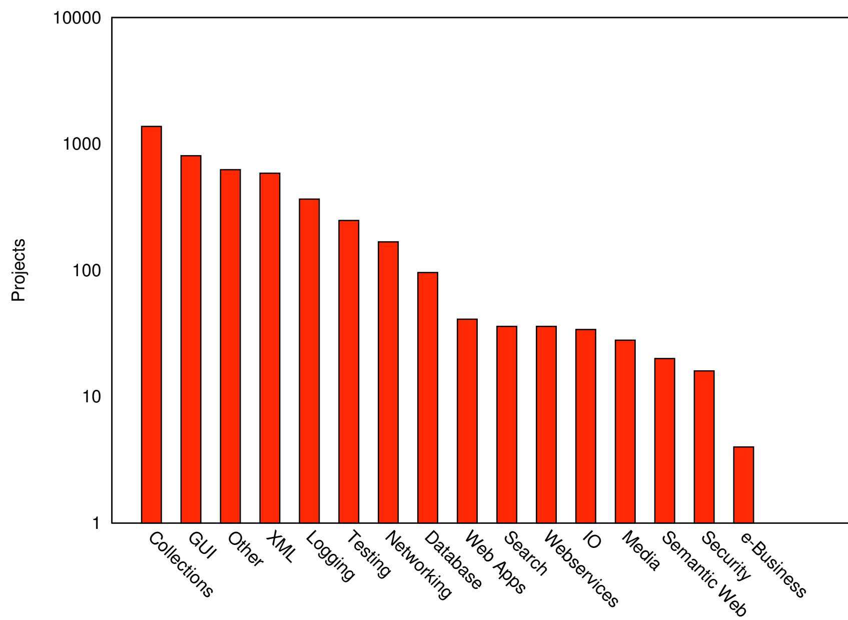


(b) Number of projects referencing an API (ordering as above)

Figure 4.2: Popularity of the 25 most referenced APIs



(a) Number of feature references to a domain



(b) Number of projects referencing a domain

Figure 4.3: Popularity of the APIs' domains

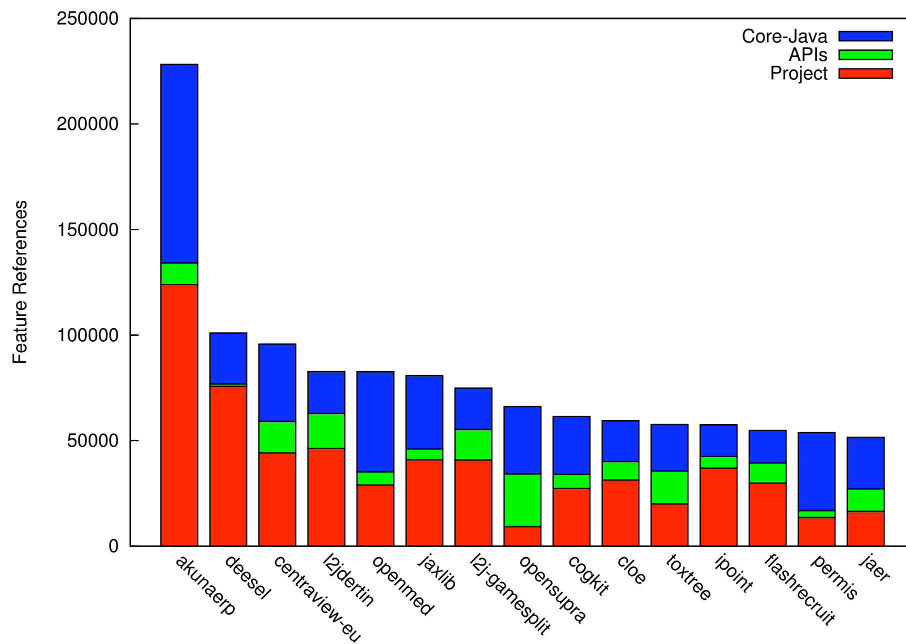


Figure 4.4: The distribution of features from Core Java, 3rd party APIs (including automatically detected ones) and project code in the largest projects

size. It becomes evident that the maximum number of packages used is directly proportional to the projects' size (the graphs use a logarithmically scaled abscissa, however, because the large differences in project size would make a linearly scaled plot hard to read).

The fact that some projects with less than 100 feature references have seemingly far too many packages for so little code can be attributed to both coding problems (like incomplete pre-alpha software with skeleton code in a complex package structure showing up in the corpus) and to errors during the automated build process.

4.3 API Usage

Figure 4.7 gives an overview of the amount of the percentage of API code in the projects from our corpus. The plot gives the number of features in each project on the abscissa, while the ordinate represents the ratio of API or Core Java features to all feature references in the project, with a ratio of 1 indicating that a project uses only API or Core Java features.

The ratio shows no clear correlation to the projects' size. On the contrary, we notice that both very small and very large projects exhibit ratios of up to 1.0.

Those data points that are drawn in red represent projects from the control group (for details on control group generation, see chapter 3.6, p. 24). Their distribution among the other data points shows that the great variation in ratios, and the seeming independence of ratio and project size, are also found among stable projects.

This graph combines API and Core Java features in one category. In order to determine whether Core API usage dominates third party API usage, we measured the ratio between these two groups. Figure 4.8 shows the result.

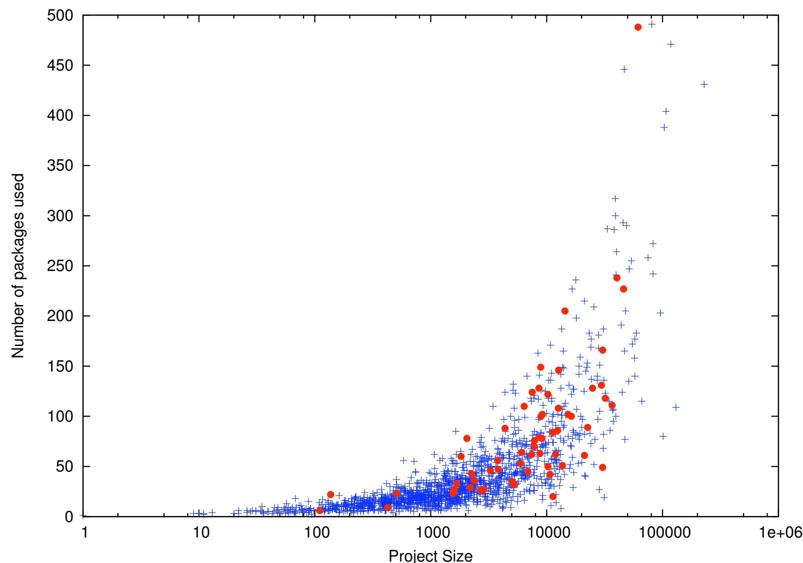


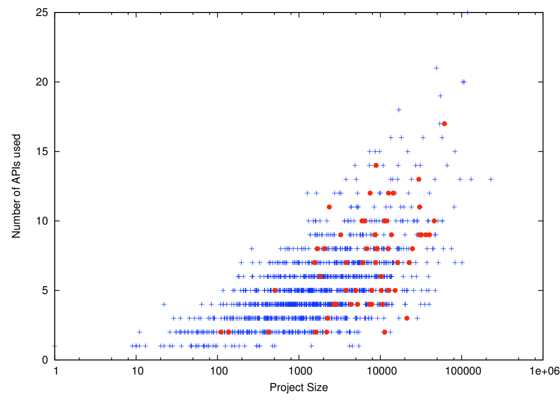
Figure 4.5: The number of packages used in each project, plotted in relation to the projects' size, measured by the number of feature references. Reference projects are drawn as red dots.

A look at the numeric values behind these measurements, given in table 4.4 on page 42, confirms this observation. The tables give mean and median values for the percentage of API code in projects, split up into categories by the projects' maturity rating (compare subsection 3.6.1 on page 26. In these tables, level 0 corresponds to projects without a self-assessment).

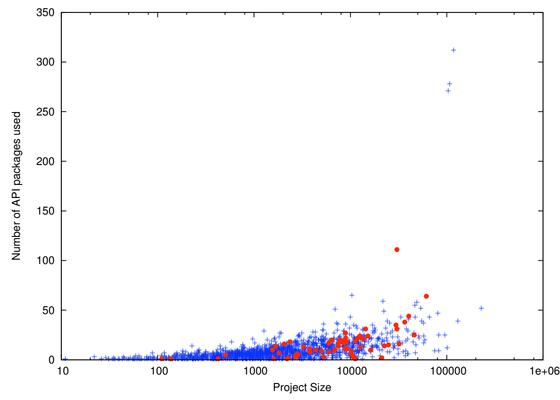
Table 4.4(a) combines 3rd party and Core Java features for the API percentage, while table 4.4(b) only gives values for 3rd party APIs. This is also the reason for the lower number of projects in the latter: Some projects do not use 3rd party APIs at all and, thus, do not show up in the base data for that measurement. We note that

- Core Java APIs dominate 3rd party API usage by far and
- the percentage of API features in a program does not increase markedly with the perceived maturity of the code.

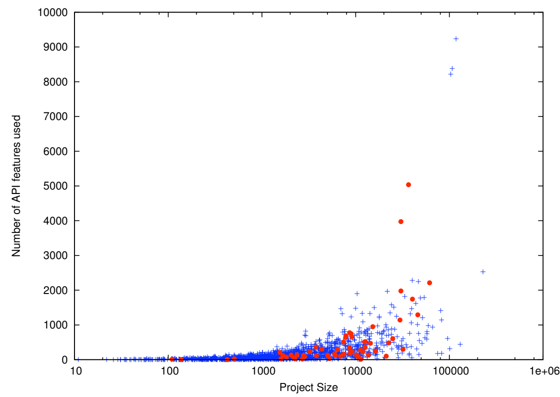
As for all measurements that rely on data from the AST-based fact extractor, there are systematic errors in the data aquisition process. These errors are discussed in section 2.6. In order to get an idea of the situation in those projects that did not compile successfully and hence do not show up in the above plots, we did a very basic approximation of how many APIs are used by each project in the entire corpus that is based on data gathered by the lexeme-based fact extractor. Figure 4.9 shows a graph of this dataset. The API count was derived from the number of different characteristic package name prefixes that were discovered in the `import` statements of a project's source files. This measurement, hence, counts only tagged APIs as we collected package prefixes only for that set. There is one systematic error remaining: We found that some of the largest "projects" showing up in this dataset are actually rather small, but their developers check in multiple versions in parallel so that a simple line count, which was used as a proxy for project size here, overestimates the project size by a factor n for n versions that



(a) Distinct APIs



(b) Packages from known APIs



(c) Features from known APIs

Figure 4.6: For different levels of granularity, these plots show how much API code is used in each project, in relation to the project's size. Reference projects are drawn as red dots. These plots show only manually tagged APIs and hence do not include features that were tagged by the automated API detection mechanism described in section 3.5.

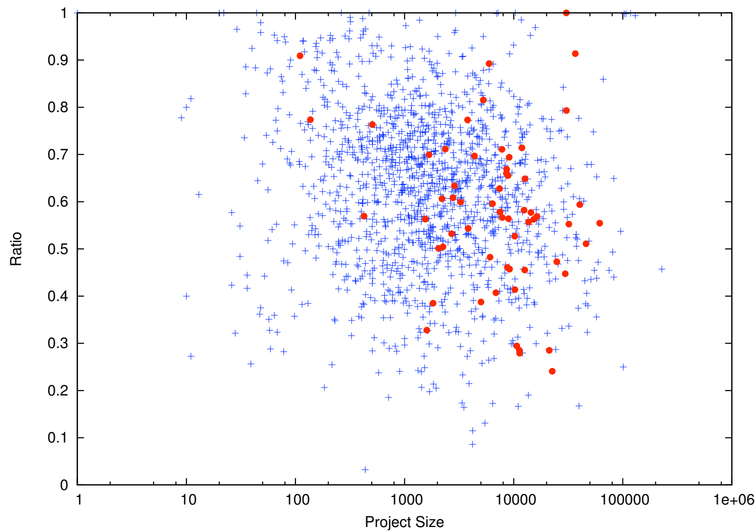


Figure 4.7: This plot shows the ratio of API features in a project in relation to the project's size. Reference projects are drawn as red dots. The ratio is computed as $\frac{|FR_A|}{|FR|}$.

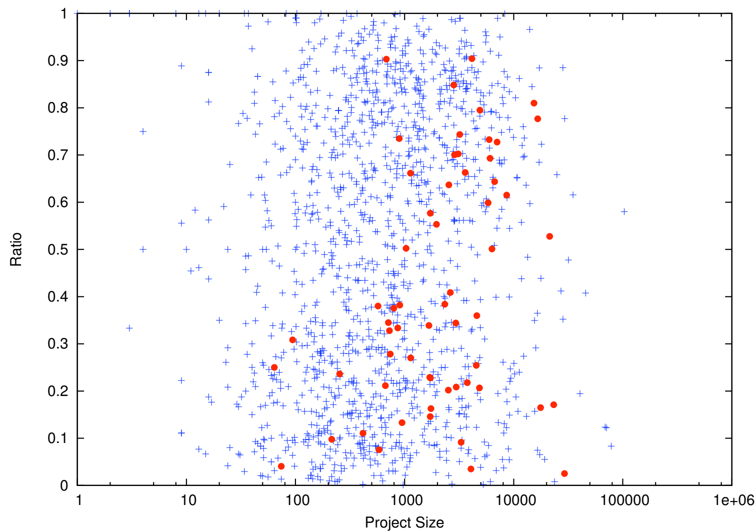


Figure 4.8: The ratio of calls to Core Java APIs compared to all calls to known APIs, plotted in relation to project size on a logarithmic scale. Again, reference projects are plotted as red dots. A value of 1 indicates that only API features from Core Java (like Swing or the Collections API) are used.

Table 4.4: Ratio of API code in projects of different maturity levels. (a) with Core Java APIs and 3rd party APIs combined, (b) only with 3rd party APIs.

Maturity Level	0	1	2	3	4	5	6	7
Projects	79	205	182	325	383	273	16	13
Average	0,6496	0,6426	0,6382	0,6061	0,6218	0,6100	0,6181	0,6445
Median	0,6633	0,6541	0,6474	0,6004	0,6441	0,6094	0,6072	0,6386

(a)

Maturity Level	0	1	2	3	4	5	6	7
Projects	70	178	158	292	356	244	16	12
Average	0,1022	0,1117	0,1212	0,1241	0,1173	0,1059	0,0901	0,1461
Median	0,0513	0,0652	0,0770	0,0704	0,0632	0,0513	0,0636	0,0285

(b)

are checked in in parallel. For example, the wingS project[1] has a repository at SourceForge[2] which contains code for three project versions. When checking out the code from the repository, this becomes obvious by investigating the first-level directories wings1, wings2 and wings3, each containing a source tree.

4.4 Feature Usage Frequency Distribution

4.4.1 In the entire corpus

Because of the structure of most APIs, which require creating and parametrizing objects in a similar manner for each application, it stands to reason that some features will be used more often than others. In order to check this assumption, we measured how frequently program and API features were referenced. Figure 4.10 shows the result of this measurement.

We note that, while there are very few features with many references, the vast majority of features is referenced only rarely. In fact, only 86 distinct features are referenced more than 10000 times, while 298920 distinct features (out of a total of 304435) are referenced less than 100 times.

The plot suggests a Zipf-style distribution, with the usage frequency of the nth feature being roughly proportional to $c \cdot n^{-1}$.

In [Vel05], the author presents measurements that suggest a Zipfian distribution for feature usage frequencies in a similar corpus-based study. His data soures were compiled programs from three major Unix-like operating system distributions.

He defines an entropy parameter $H \mid 0 \leq H \leq 1$ for programs in given problem domains, showing that for domains with maximum entropy $H = 1$ (where each program differs from each other possible program), libraries may not be used to “shorten” programs (i.e., to reduce the amount of code the developer has to write), whereas for realistic problem domains with $0 < H < 1$, there is a potential for code reuse with the help of libraries. The reason for the observed Zipfian distribution, he argues, is “a direct result of programmers trying to write as

[1] http://www.wingsframework.org/
[2] https://j-wings.svn.sourceforge.net/svnroot/j-wings/

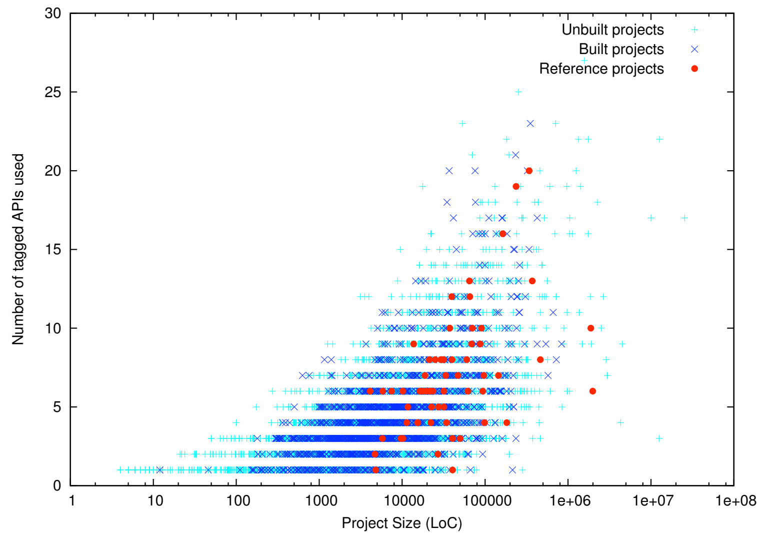


Figure 4.9: An approximation of the number of distinct APIs used by each project in the entire corpus, created based on data from the lexeme-based scanner. Project size is measured in lines of code.

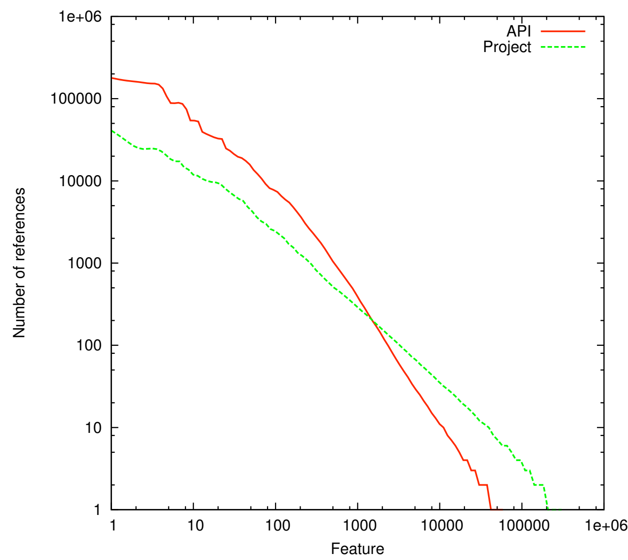


Figure 4.10: The feature usage frequency distribution for API and project features. Features are ordered by the number of references to them.

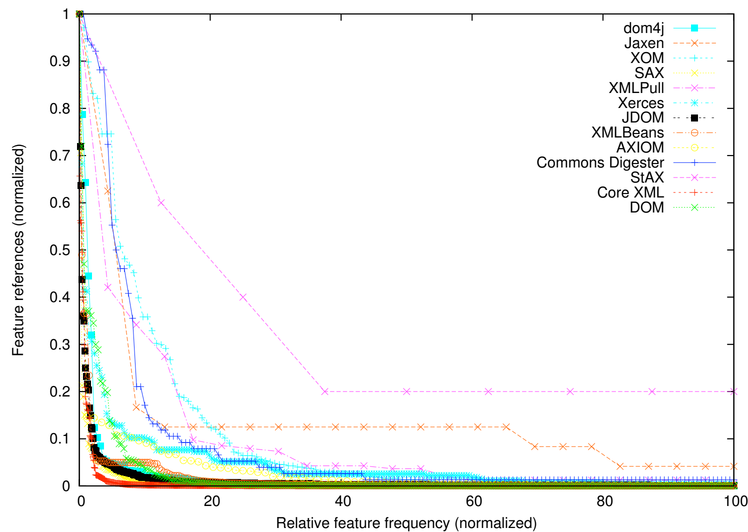


Figure 4.11: The feature usage distribution for APIs in the “XML” domain.

little code as possible by reusing library subroutines; this drives reuse rates toward a ’maximum entropy’ configuration, namely a Zipf’s law curve” [Vel05, § 1].

If we plot the uses of API and project features as two distinct graphs, we get the plot shown in fig. 4.10. While Veldhuizen’s observation seems to hold for both API and project features, we observe that the API curve is sloped slightly more steeply. This indicates that there are fewer seldom-used features in API code than in project code.

Intuitively, one may think that a curve with an inverted-L-shape (corresponding to a lot of often-used features and very few seldom-used features) would indicate maximum API usage efficiency, because API writers would then not have spent time and effort on the development of seldom-used features. All entropy-generating code, the features that differentiate programs from each other and are less suitable to be considered for code reuse, would be left to application developers. However, Veldhuizens work states that for overall entropy maximization (and, accordingly, minimization of writing effort for client programmers), both curves should be Zipfian, which is just what we observe.

4.4.2 In APIs

If we take a more detailed look at how usage frequencies are distributed for API features, the general impression of feature usage following tail-heavy distributions remains. Figure 4.11 shows relative feature usage statistics for APIs in the “XML” domain (this constraint was applied for readability reasons only and does not significantly change the trend visible in the chart). Because the absolute usage numbers differ greatly between the APIs, the values are given as percentages relative to the most often used feature. Again, we note that there are very few heavily used features and a “long tail” of rarely used ones.

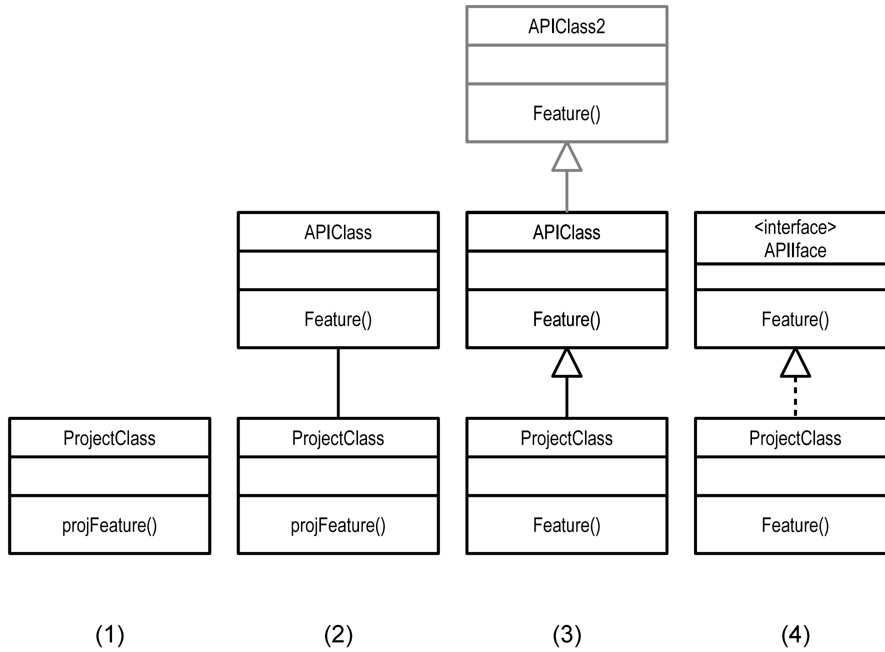


Figure 4.12: Inheritance scenarios in the context of API coverage

4.5 Coverage of API Features

For applications in automated API Migration, it is interesting to consider the percentage of features that are used at least once in the corpus. We call this number the *coverage* of an API's features.

Depending on the intended application, there are several ways of measuring this value. The different approaches we took are illustrated by the UML diagram in fig. 4.12. Subfigure (1) in that diagram refers to simply counting feature references in project code, as was done above.

If we extend this simple measurement to reflect API usage, we may just count the different methods and constructors that are called at least once in some project. This straightforward method, shown in subfigure (2), covers the vast majority of API usage we see in our corpus.

One may extend this notion of API coverage to include features that are used transitively through inheritance from and implementation of API-provided features. Subfigure (3) illustrates what, in the result graph below, is refered to as "transitive feature usage": project classes may extend API classes (that may themselves inherit from others, as hinted at by the grayed-out class), and all through this inheritance tree, we count each occurrence of the feature that is used in project code as "used". This view is useful when one needs to consider the scope of changes to a feature somewhere in the inheritance tree, but there are only very few instances where API classes are extended in this fashion.

Implementation of interfaces, as shown in subfigure (4), is also analyzed. These use cases are shown under the label "implementation" in the results figure.

While these measurements focus on the number of different API features used, it is also interesting to consider the absolute usage figures for feature implementation. Table 4.5 gives these values for cases where project classes implement features from API code.

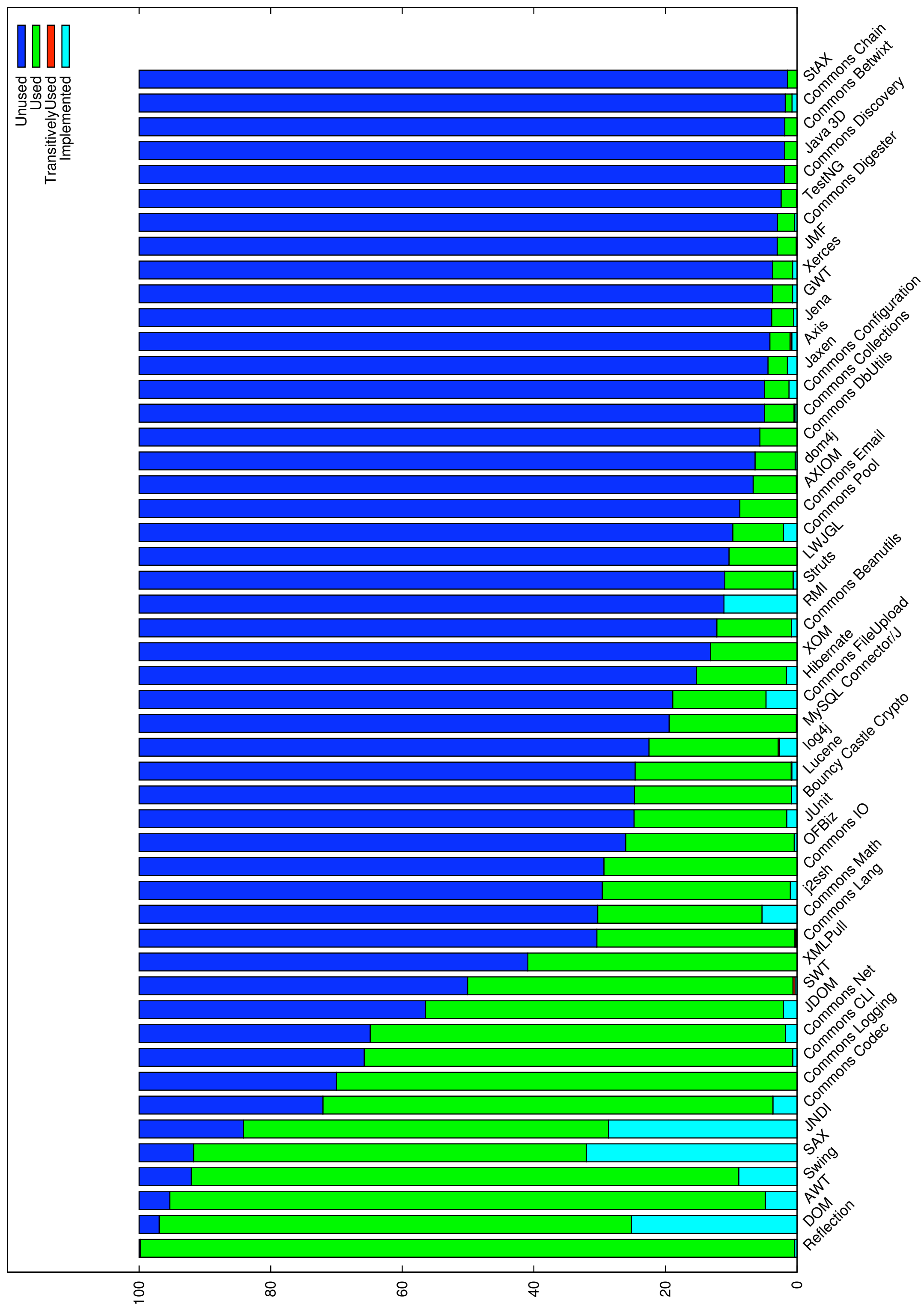


Figure 4.13: This bar chart shows the percentages of API features that are used directly, via inheritance and via implementation. Because of space constraints, it was limited to the 50 APIs with the highest coverage values.

API	Implemented Features
Swing	11150
AWT	756
JUnit	344
Xerces	143
GWT	143
SAX	90
log4j	87
SWT	86
Lucene	22
Struts	14
Core XML	13
jMock2	7
JMF	6
GNU Trove	5
Commons Lang	4
JFace	4
Axis	4
Commons Net	3
Commons Pool	3
Commons Digester	2
Bouncy Castle Crypto	2
LWJGL	1
RMI	1
Commons Beanutils	1
Berkeley DB	1
Commons DbUtils	1
Hibernate	1

Table 4.5: Implementations of API features

5 Related Work

There have been a lot of publications in fields closely related to our work, however, the analysis of API usage from large corpora has seldom been discussed. Hence, we split the discussion of related work into sections corresponding with the topics that we needed to deal with in our work.

5.1 Structure of Real-World Programs

There are several projects aiming at analysing the structural properties of software projects.

In [BFN+06], the authors present an analysis of the structure of 56 manually selected Java programs. The study avoids the problems associated with selecting SourceForge as the only data source, as discussed in section 2.6.1, by using projects from six different repositories, as well as using only mature, actively developed projects. The authors used the byte code of the projects in the corpus for the analyses, which means they were unable to access type information. The study found power-law relations for some of the metrics that were measured, but not for all.

Another search for power-law distributions in software metrics was presented in [WC03]. This analysis focuses on the internal structure of classed and finds, amongst other results, that the *number* of certain software features like subclasses, methods etc. in classed follows power-laws. This analysis was performed on a corpus of only three high-profile projects, namely the JDK, Apache Ant and Tomcat.

[TDX07] discusses methods for the detection of changes in APIs. The authors argue that, while the API to a software library should remain stable if possible, refactorings tend to cause changes to the API. Tools like this may be a helpful addition to our system because we currently do not have a means of dealing with different API versions that may have changed since some of the older projects in our code were written.

5.2 Simple Project Statistics

We use some of the statistics provided by SourceForge for the selection of a set of reference projects (see chapter 3.6). The data for this analysis can be gathered by simply processing the HTML code of SourceForge's project pages. This method was used by several authors to create quantitative analyses of large corpora.

One early discussion of the method and the available data is presented in [HK05].

Weiss ([Wei05a, Wei05b]) uses it to analyze project maturity levels, the distribution of developers across projects and several other metadata provided by SourceForge, like licenses and the "intended audience" entries. He notices that Java, together with C and C++, is one of the dominant languages for SourceForge's projects, and that there are two factors that complicate analyses like ours: Project metadata is dynamic and changes often, and there are many projects that rate themselves as "unstable" or "in planning". Hence, analyses will generaly only represent a snapshot of a short timeframe, which can be extended by using a control group of known stable projects. Unstable projects are not suitable for code analyses because of the large probability of "hacks" in the code that the developers will correct later. We have tried to assess the impact of this by defining a set of reference projects.

In [Kri02], the author uses similar analysis methods to show that most projects hosted on SourceForge are developed and maintained by only very few people.

The FLOSSmole project ([HCC06], [Flo]) offers statistical data from several open source project hosting websites in a pre-processed form. For simple analyses, its database dumps may save users the time and effort involved in HTML-scraping the original web sites, while at the same time reducing the impact on SourceForge's servers.

FOSSology, described in [Gob08], is a project that develops a framework for software analyses. It is currently focusing on checking the software licenses used in software systems, with additional modules providing some metadata analysis capability, but can be extended to perform other tasks as well.

5.3 Clone detection

While detection of code clones is not a centerpiece of our work, but just a necessary step in preparation of the raw data for analysis, we have to rely on methods that are not discussed in the litarature. As discussed in section 3.7, we need to identify cases where API code is included in other APIs as well as cases where API code is included directly in program code. Most papers on software clone detection, however, focus on program-internal clones and reusability considerations.

[RCK09] provides a comprehensive overview of the field of software clone detection.

One of the projects that might be adapted for the task of finding the sort of code clones we encountered here is D-CCFinder by Livieri et al. [LHMI07], which is based on CCFinder [KKI02].

5.4 API Usage

5.4.1 Theoretical Considerations

The degree of reuse of software components is an important factor in assessing API usage. Veldhuizen provides an important work on software reuse ([Vel05]), which discusses the degree of possible reuse and its dependency on application domains. He provides boundaries for the possible degrees of code reuse in application domains, arguing that this value is an intrinsic property of the domain and generally independent of external factors. For an application of his findings to our work, see chapter 4.4.

5.4.2 Mining Frequent Usage Patterns

There are several tools and projects that aim to provide developers with API usage examples from real-world code, or to understand how a certain API is used "in the wild". One main analyis method used in this area is the search for frequent sequences of feature references, *frequent pattern mining*. Often-used patterns are generally taken as examples of coding practices. The analysis methods used often share some characteristics with our approach in that they evaluate API usage in a (usually rather small) corpus of program code and provide statistical analyses based on the gathered data. However, the analysis method we use in this work is by design unable to analyse feature call sequences, limiting us to analysing quantitative metrics.

An early but thorough work on practical code reuse analysis is [Mic00], introducing A. Michail's CodeWeb tool. He analyses code reuse in the context of the KDE application framework, which, in 2000, was purely C++. The tool goes beyond a simple frequency analysis of feature usage by providing some pattern analysis based on the usage of API features. The results of this analysis,

which are of the general form "if KApplication is instantiated, a call to KApplication::exec() follows in 72.3% of cases", can serve to inform API developers about feature usage distribution, but Michail does not aggregate the results across classes. An advantage of the analysis technique used is the ability to correctly treat inheritance.

The MAPO project [XP06] uses analyses of large code collections to provide developers with examples of API usage. The system queries several code search engines for the feature given and processes their result sets. It does, in contrast to our work, not aim to provide large-scale usage statistics, but instead focuses on providing usage examples for specific API features. The authors intend to continue their work with the goal of mining frequent usage patterns and providing searchers with synthesized sample code.

Thummalapenta et al. present an analysis of the feature usage distribution in APIs, with the goal of finding often-used sets of features. After calculating call statistics for API features in a given program, the SpotWeb tool presented in [TX08] bins and clusters usage data, returning sets of features that are most often used, called "hotspots" by the authors. By detecting dependencies between hotspots, the tool is able to present examples of how a feature is commonly used. This shows a possible extension of the analyses on our data: By doing analyses on the distribution of n-grams of feature references in project classes, one may be able to provide hints to hotspots in our corpus.

This sort of analysis has been done in several projects, differing in the representation of usage data.

Ammons et al. [ABL02] analyze program execution traces in order to provide a *finite-state-automaton* representation of API properties. Their work aims at automated mining of specifications for software verification from frequent execution traces.

The PR-Miner project [LZ05a] represents its results as *frequent itemsets*, disregarding the order of feature references.

[AXPX07] describes an approach that uses the extraction of usage scenarios from a code base to create specifications for use in a model checker. The authors use *partial orders* of features as a representation of sequences of feature references. The specifications obtained by code mining are intended to be used for assisting code quality reviews.

Other projects also try to use the data gathered by frequent pattern mining for quality improvement methods. [LZ05b] discusses the DynaMine tool, which searches for violations of coding rules in applications. As DynaMine also uses version history mining, it is somewhat beyond the scope of this work. Both [WN05] and [AX09] are related papers, discussing the mining of usage patterns from code repositories and applications in handling software errors.

A general overview of the field of frequent pattern mining, with pointers to common algorithms and a short section on applications in software analysis, is given in [HCXY07]. Buehrer et al. [BPG06] provide a comparison of implementations and suggest optimizations for frequent itemset mining in large datasets. Another work presenting efficient methods for software mining applications is [GZ03].

6 Further work

Given the limited scope of this work, several analyses leave room for further studies.

The "code clone" phenomenon as described in 3.7.1 and violations of Java's package naming conventions have only been discussed superficially here, and should be investigated in greater detail before applying our analysis method to further analyses.

In the originally projected context of research into API migration, it is also interesting to consider the development of project and API code over time, an aspect we still neglect in this work. While our data storage structure can correctly represent programs built with different Java versions, we ignore the versioning information that is available from project repositories and sources of binary API distributions.

Additionally, there remain several possible improvements to the analysis processes described in this thesis that would be worth implementing.

First and perhaps foremost, every statistical analysis of real-world programs benefits from a large sample base containing a set of known high quality projects. However, the corpus we created for this thesis was rather small, compared with the large number of projects that are available on SourceForge and elsewhere. The reasons for this were discussed in section 2.3. Hence, support for more different build mechanisms, especially GNU `make`, and using more code repositories as data sources should lead to results that better represent the state of open source Java development. This would also help mitigate the effects of some of the concerns laid out in section 2.6, namely the systematic loss of projects that are hosted and / or developed differently.

As our analysis method requires access to the projects' source code, it was easiest for us to concentrate on a corpus of open source software. Given the low figures for breadth and depth of API usage we found, it might be interesting to compare the metrics of an open-source corpus like ours to a corpus of professionally developed closed-source software.

Another area of improvements is the structure of the analysis environment. For some queries, a relational database does not provide the most natural representation of object-oriented software components, making e.g. recursive descents into inheritance hierarchies unnecessarily difficult. An object-oriented database may be better suited to at least some tasks. Alternatively, there are several domain-specific query languages that might be adapted to our database, some of which are compared in [AR08]. Apart from the database itself, the Python scripts used for querying the database and processing the results offer room for improvements, for example by offering an interactive query interface.

6.1 Alternative approaches

Apart from improving the presented method of analysing data gathered from an instrumented compiler, it may also be worthwile to investigate alternative data sources.

For working with Java programs, the Byte Code Engineering Library (BCEL) from the Apache Jakarta Project[1] or similar bytecode engineering tools may offer an approach to extracting feature usage information from existing binary code. However, while this may be the only method to make software that is not available as source code available for analysis, there is some information

[1] `http://jakarta.apache.org/bcel/`

that would be inaccessible: Mainly commented-out code, unused packages like sample code and comments. Whether or not this poses a problem will depend on the questions the analyses try to address.

7 Conclusion

We have presented an analysis environment that allows flexible observations of API usage in large code corpora. Like many other corpus-based studies of real-world programs, we use SourceForge as our main data source. Our mechanisms for project selection, downloading and collecting project statistics from the SourceForge repository are similar to what several other projects have done before (compare section 5.2).

The AST-based measurements we heave taken demonstrate an approach for a pragmatic large-scale analysis of API usage in a software corpus. Instrumenting the Java compiler with a compiler plugin that writes information from the AST out into a database, and using build tools like ANT that allow a high grade of automation, allows running analyses with very little manual intervention. The most time-consuming manual steps remaining are tagging of APIs and the selection of packages that are delivered with the API but do not themselves provide API features.

With regard to the measurements, our corpus seems to show a tendency towards few and "shallow" use of APIs. Most projects in the corpus use only few features from APIs, and the APIs we examine are not used exhaustively. For the projects in the control group, consisting of mature projects, the percentage of features that stem from APIs is markedly higher.

Bibliography

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002. doi:http://doi.acm.org/10.1145/565816.503275.

[Abs] Java Platform SE 6: Specification of AbstractProcessor [online]. Available from: http://java.sun.com/javase/6/docs/api/javax/annotation/processing/AbstractProcessor.html [cited Nov 7, 2010].

[AR08] Thiago L. Alves and Peter Rademaker. Evaluation of code query technologies for industrial use. In *IEEE International Conference on Program Comprehension*. Royal Netherlands Academy of Arts and Sciences in Amsterdam, 2008.

[AX09] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, pages 370–384, March 2009. Available from: http://www.csc.ncsu.edu/faculty/xie/publications/fase09.pdf.

[AXPX07] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, New York, NY, USA, 2007. ACM. doi:http://doi.acm.org/10.1145/1287624.1287630.

[Bea] Beautifulsoup web page [online]. Available from: http://www.crummy.com/software/BeautifulSoup/ [cited Nov 7, 2010].

[BFN+06] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the shape of java software. *SIGPLAN Not.*, 41(10):397–412, 2006. doi:http://doi.acm.org/10.1145/1167515.1167507.

[BPG06] Gregory Buehrer, Srinivasan Parthasarathy, and Amol Ghoting. Out-of-core frequent pattern mining on a commodity PC. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 86–95, New York, NY, USA, 2006. ACM. doi:http://doi.acm.org/10.1145/1150402.1150416.

[Cla] The java virtual machine specification, §5.3: Creation and loading [online]. Available from: http://java.sun.com/docs/books/jvms/second_edition/html/ConstantPool.doc.html#72007 [cited Nov 7, 2010].

[CMS07] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of java bytecode programs. *Softw. Pract. Exper.*, 37(6):581–641, 2007. doi:http://dx.doi.org/10.1002/spe.v37:6.

[Dar] Joe Darcy. Java Specification Request 269: Pluggable annotation processing API. online. Available from: http://jcp.org/en/jsr/summary?id=269.

[Flo] Flossmole – collaborative collection and analysis of free/libre/open source project data [online]. Available from: http://ossmole.sourceforge.net/ [cited Nov 7, 2010].

[Gob08] Robert Gobeille. The FOSSology project. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 47–50, New York, NY, USA, 2008. ACM. doi:http://doi.acm.org/10.1145/1370750.1370763.

[GZ03] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[HC04] James Howison and Kevin Crowston. The perils and pitfalls of mining sourceforge. In *In Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, pages 7–11, 2004.

[HCC06] James Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[HCXY07] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data mining and Knowledge Discovery*, 15(1):55–86, Jan 2007.

[HK05] Michael Hahsler and Stefan Koch. Discussion of a large-scale open source data collection methodology. *Hawaii International Conference on System Sciences*, 7:197b, 2005. doi:http://doi.ieeecomputersociety.org/10.1109/HICSS.2005.204.

[java] Java Development Kit 6: Java compiler (javac)-related APIs & developer guides [online]. Available from: http://java.sun.com/javase/6/docs/technotes/guides/javac/index.html [cited Nov 7, 2010].

[Javb] Java Platform, Standard Edition 6 API Specification [online]. Available from: http://java.sun.com/javase/6/docs/api/ [cited Nov 7, 2010].

[Javc] Why developers should not write programs that call 'sun' packages [online]. Available from: http://java.sun.com/products/jdk/faq/faq-sun-packages.html [cited Nov 7, 2010].

[JDK] JDK and JRE file structure [online]. Available from: http://java.sun.com/javase/6/docs/technotes/tools/solaris/jdkfiles.html [cited Nov 7, 2010].

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1019480.

[Kri02] Sandeep Krishnamurthy. Cave or community? an empirical examination of 100 mature open source projects. *First Monday [Online]*, 7 (6), 2002.

[LHMI07] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed

CCFinder: D-CCFinder. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society. doi:http://dx.doi.org/10.1109/ICSE.2007.97.

[LZ05a] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM. doi:http://doi.acm.org/10.1145/1081706.1081755.

[LZ05b] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005. doi:http://doi.acm.org/10.1145/1095430.1081754.

[Mic00] Amir Michail. Data mining library reuse patterns using generalized association rules. In *ICSE*, pages 167–176, 2000. doi:http://doi.acm.org/10.1145/337180.337200.

[RCK09] Chancal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[Sun] Java SE Documentation: Annotated outline of collections framework [online]. Available from: http://java.sun.com/javase/6/docs/technotes/guides/collections/reference.html [cited Nov 7, 2010].

[TDX07] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 377–380, November 2007. Available from: http://www.csc.ncsu.edu/faculty/xie/publications/ase07-refaclib.pdf.

[TX08] Suresh Thummalapenta and Tao Xie. Spotweb: detecting framework hotspots via mining open source repositories on the web. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 109–112, New York, NY, USA, 2008. ACM. doi:http://doi.acm.org/10.1145/1370750.1370775.

[Vel05] Todd L. Veldhuizen. Software libraries and their reuse: Entropy, kolmogorov complexity, and zipf's law. *CoRR*, abs/cs/0508023, 2005.

[WC03] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. *CoRR*, cs.SE/0305037, 2003.

[Wei05a] Dawid Weiss. A large crawl and quantitative analysis of open source projects hosted on sourceforge. In *Poznań University of Technology Technical Report RA-001/05*, 2005.

[Wei05b] Dawid Weiss. Quantitative analysis of open source projects on sourceforge. In Marco Scotto and Giancarlo Succi, editors, *Proceedings of the First International Conference on Open Source Systems, Genova*, pages 140–147, 2005.

[WN05] W. Weimer and G.C. Necula. Mining temporal specifications for error detection. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005.

[XP06] Tao Xie and Jian Pei. Mapo: mining api usages from open source repositories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57, New York, NY, USA, 2006. ACM. doi:http://doi.acm.org/10.1145/1137983.1137997.