Delft University of Technology
Software Engineering Research Group
Technical Report Series

The maven repository dataset of metrics, changes, and dependencies

Steven Raemaekers, Arie van Deursen, and Joost Visser

Report TUD-SERG-2013-005

TUD-SERG-2013-005

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the 10th Working Conference on Mining Software Repositories 2013, IEEE. http://dl.acm.org/citation.cfm?id=2487129

The Maven Repository Dataset of Metrics, Changes and Dependencies

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

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

***Abstract*—We present the Maven Dependency Dataset (MDD), containing metrics, changes and dependencies of 148,253 jar files. Metrics and changes have been calculated at the level of individual methods, classes and packages of multiple library versions. A complete call graph is also presented which includes call, inheritance, containment and historical relationships between all units of the entire repository. In this paper, we describe our dataset and the methodology used to obtain it. We present different conceptual views of MDD and we also describe limitations and data quality issues that researchers using this data should be aware of.**

***Index Terms*—Maven repository, Dataset, Data mining**

I. Introduction

We present the Maven Dependency Dataset (MDD), which contains metrics, changes and dependencies of 148,253 jar files. The goal of this dataset is to facilitate replicable large-scale research on software releases, versions and evolving dependencies at the level of packages, classes and methods. MDD contains code metrics, dependencies, breaking changes between library versions and a complete call graph of the entire Maven repository. This makes it possible to answer a wide range of software evolution-related research questions, such as the following:

- Can we predict when code changes will occur?
- Can we estimate the impact of these changes?
- How fast do libraries adapt to changes in dependencies?
- What patterns can we observe in changes of methods, packages and classes?
- What code properties are associated with a high adaptation and survival rate of library versions?
- How can we visualize library evolution through time?

MDD facilitates answering these and other research questions and we therefore invite other researchers to explore our dataset and use it in innovative ways.

Source and binary jar files from the Maven repository[1] were used to fill the dataset, which is an online repository of open source Java libraries. Software developers using Maven can fully specify the entire build process of a software library in a single configuration file, including required third-party libraries. When building a project, Maven automatically downloads specified dependencies from a specified repository. The system is mostly used for Java programs, but can be used for other languages and even non-source code artifacts as well. The mechanism addresses frequently occurring problems of missing dependencies and compilation errors when rebuilding software written on other developers' machines.

We enriched the Maven dataset with a set of evolution-related metrics to answer research questions about software evolution and maintenance. The size of the dataset and the fact that a large number of different development teams have been releasing artifacts over a large timespan makes it a valuable source for data analysis and hypotheses testing in the field of software evolution. Collected data includes size information (e.g. LOC, number of methods), evolution information (e.g. number of removed methods per release, breaking changes per release) and a complete call graph of the entire repository, containing four different types of dependencies: containment, historical, call and extension/inheritance.

This paper is structured as follows. In Section II, the permanent download location of our dataset can be found. In Section III, descriptive statistics are presented. Section IV presents the data schemas of databases in our dataset. In Section V, our data collection approach is outlined. In Section VI, data quality issues and limitations of our dataset are discussed.

II. Download location

The accompanying website for this paper, containing a detailed per-column description of the dataset, an addendum to this article and installation instructions for the dataset can be found at the following location:

`http://www.sig.eu/en/msr2013b`

The dataset itself can be found at the following location:

`dx.doi.org/10.4121/uuid:68a0e837-4fda-407a-949e-a159546e67b6`

III. Descriptive statistics

We used a snapshot of the central repository dated July 30, 2011. Descriptive statistics of the dataset can be found in Figure 1. As can be seen in the upper table, the dataset contains a total of 148,253 jar files. When uploading a library to the central repository, library developers can upload binary, source and javadoc jars. Note that not all library versions are uploaded

[1] http://search.maven.org

with corresponding source and javadoc jars: only 101,413 of 148,253 libraries (68.4%) have source code available and only 78,766 libraries (53.1%) have javadoc available.

The second part of Figure 1 gives information on the size of libraries. It shows that the 75th percentile of number of lines of code is at 2,200, indicating that most libraries in the repository are relatively small. There are 22,111 artifacts (projects) in the repository, with on average 6.7 versions per artifact.

Number of binary jar files	148,253
Number of source jar files	101,413
Number of javadoc jar files	78,766
Unresolved jar references*	3,319
Total SLOC	350,571,247
Number of classes	4,174,150
Number of methods	37,406,546

	min	*p5*	*p25*	*p50*	*p75*	*p95*	*max*	*avg*	*sd*
loc	1.0	39.0	203	650	2.2k	17.5k	382k	4.4k	15.7k
m/j	1.0	4.0	21.0	69.0	240	1.5k	56k	468	1.7k
c/j	1.0	1.0	3.0	10.0	30.0	223	4.7k	52.23	166.7
d/j	1.0	1.0	2.0	5.0	8.0	18.0	211	6.5	7.02
v/a	1.0	1.0	1.0	3.0	7.0	26.0	383	6.7	12.24
a/g	1.0	1.0	1.0	2.0	4.0	19.0	306.0	4.87	12.23

Fig. 1. Descriptive statitistics for libraries in the Maven repository. loc = lines of code, m/j = number of methods per jar, c/j = number of classes per jar, d/j = number of dependencies per jar, v/a = number of versions per artifact, a/g, number of artifacts per groupId. *Libraries sometimes refer to artifacts or versions that are not present in our snapshot.

IV. Data Schemas

For performance reasons we used three different types of database formats: a MySQL database, a Berkeley DB database and a Neo4j graph database. The graph database is most suitable to query graph-like structures such as call graphs. The Berkeley DB database is an on-disk key-value store which can look up metrics very quickly. We give a conceptual model of each of these databases in this section.

A. MySQL database

The data schema of the MySQL database is presented in Figure 2. As can be seen in this figure, it consists of the following tables:

`files` The files table contains information on all library versions. Metrics such as the number of methods (*nrUnits*), the number of methods compared to the next version (*nrNewUnits*) and other metrics are stored in this table. Libraries that are referenced by other libraries but which were not found in our dataset are entered in this table without a *fullName*. The files table also contains stability metrics which we defined in previous work [2]. For a more elaborate description of properties of individual files, see [3].

`stats` The stats *stats* table stores metrics such as LOC, McCabe, number of methods and number of classes for each library version. It also contains SIG star ratings, which are further described in [1].

`units` This table is not stored in MySQL but it is shown here to demonstrate that there exist (conceptual) foreign key relationships between the MySQL, Neo4j and Berkeley DB databases. Units can be complete files, packages, classes or methods, which are all stored in this table. Each unit belongs to a certain file and has a fully qualified name (the *name* field).

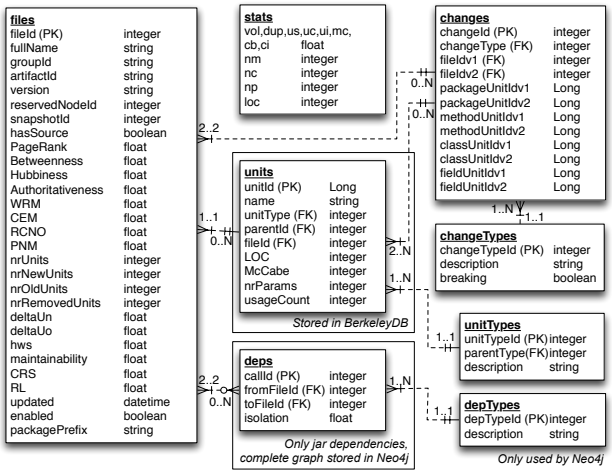

Fig. 2. The MySQL database schema. Some tables are present in the other database formats and are presented here to give an overview of the interconnection between the datasets. Foreign keys are drawn in the schema but have been removed from the database due to performance reasons; however, foreign key identifiers still match with primary key identifiers.

Metric values such as the McCabe, LOC and parameter count are also stored in this table.

`changes` Different types of changes between library versions are stored in this table. Changes can be breaking, meaning that source code has to be recompiled if using a dependency that introduces such a change. Non-breaking changes are less severe and do not require recompilation. Unit identifiers are looked up in Berkeley DB and are stored in this table, if found. In either case, names of the affected package, class, method or field are also stored for each change.

`deps` This table contains all library dependencies as present in the build configuration file of a project. When a library depends on another library, a `<dependency>` section is present in the `pom.xml` file of the project specifying the exact groupId, artifactId and version of the library it depends on. Also stored in this table is an *isolation rating*, specifying the percentage of files that does not import the dependency and is essentially a measure of encapsulation of a dependency in a system. This table only contains library dependencies; all other dependency types are stored in the Neo4j database.

Supporting tables such as `changeTypes`, `unitTypes` and `depTypes` are reference tables that give additional information on properties of changes, units and dependencies, respectively. For a complete description of all columns in the MySQL database and instructions on how to query the Berkeley DB database, see the online addendum.

B. Berkeley DB database

To make fast lookup of single methods, classes and packages possible, a Berkeley DB database was created. This database contains information on 36,695,764 different methods, classes and packages. Indices on unique unit identifiers, fully qualified name, fileId, unit type, groupId, artifactId and versions have been created to facilitate searching on any of

those fields. The unique unit identifiers match the identifiers as used in the Neo4j call graph. The fileId index refers to the fileId column in the MySQL database. Unit type is a number denoting the type of the unit: 1 = jar file, 2 = package, 3 = java file, 4 = class, 5 = method. The script `getunits.sh` in the replication package is the main interface to query the Berkeley DB database directly and can be used to obtain information on single methods, classes or packages or to obtain a list of units based on a combination of values for any of the mentioned indices.

C. Neo4j database

The Neo4j database contains all call graph information. A conceptual model is shown in Figure 3 and an example is shown in Figure 4.

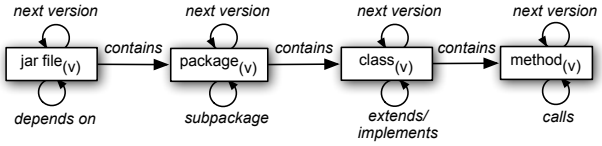

Fig. 3. A conceptual model of units in the Neo4j database. (v) = version.

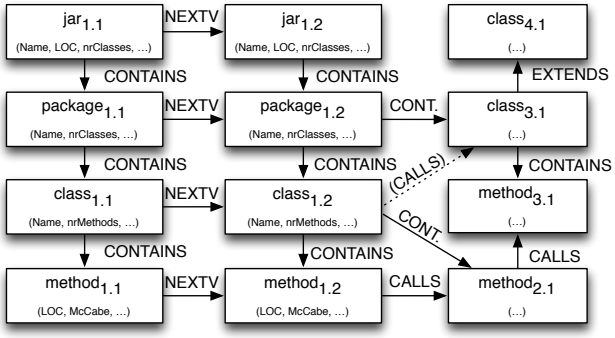

Fig. 4. An example of different versions of packages, classes, methods and their relationships in the graph database. NEXTV = next version.

The database can be queried using the Cypher query language[2]. More details can be found in the online addendum.

V. Methodology

The DAS-3 Supercomputer[3] was used to process all jar files. The Supercomputer consists of 68 dual-node 2.4 GHz computing nodes with 4 GB memory each. The system runs on ClusterVisionOS 2.1, which is based on Scientific Linux 4.3. The system has a central head node which contains the database and distributes commands to the computing nodes. The database was filled in multiple runs; each run took approximately one week. Since tasks can be easily parallellized across a large number of machines, a speedup of approximately 60 times was achieved. Without the Supercomputer, total running time was estimated to be more than one year. Special software was developed to obtain all data. Eventually, this software consisted of approximately 10,000 LOC of Java and 3,000 LOC of bash, Python and R scripts. Source code is available in the source package on the website, although exact replication of the analysis using this package is impossible due to required access to the Supercomputer. Also, source code and binaries of the SAT are not included in the package.

Figure 5 shows the steps that were taken to obtain all data.

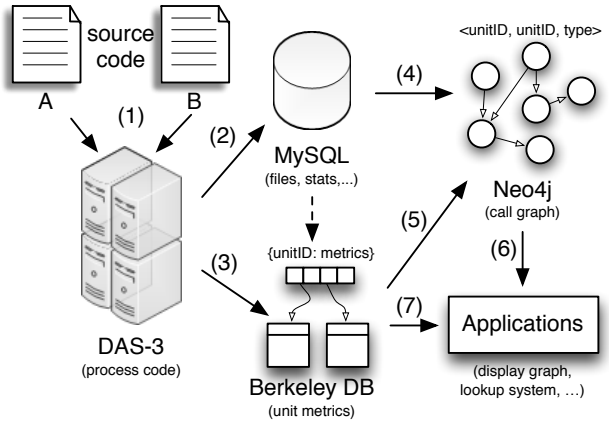

Fig. 5. An overview of the data collection approach taken in this paper.

The numbers in the figure correspond to the following steps:

(1) First, source code was processed using the Supercomputer. The Software Analysis Toolkit (SAT) of the Software Improvement Group (SIG)[4] was adapted to run in parallel in multiple machines and was used to obtain metrics and call graphs from source code.

(2) The SAT writes call graph and metric information to a MySQL database for each different artifact. We do not save all databases completely but we extract interesting information from this database and put it in a separate MySQL database.

To detect changes between library versions, we use an adapted version of Clirr[5]. This tool checks for breaking changes between each two subsequent versions of binary jar files. A breaking change is any change in the next version of a binary jar file which causes compilation errors in systems using it. These changes are also referred to as *binary incompatibilities* and require users of those libraries to adjust and recompile their code. There exist several types of breaking changes; examples are public method and class removals. The Eclipse Wiki has more information on binary (in)compatibilities in Java[6] and the Java Language Specification[7] contains formal definitions and explanations.

(3) Metrics on more than 200 million methods, classes and packages were collected. To make fast lookup possible, we stored this information in Berkeley DB, a dedicated on-disk key-value store. We created several keys to obtain information on units, such as fully qualified names, unique identifiers and

[2]http://docs.neo4j.org/chunked/stable/cypher-query-lang.html
[3]http://www.cs.vu.nl/das3
[4]http://www.sig.eu/en
[5]http://clirr.sourceforge.net
[6]http://wiki.eclipse.org/Evolving_Java-based_APIs
[7]http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html

library names. This enables fast retrieval of units that satisfy certain selection criteria.

(4) We use the obtained call graph information to build a graph of methods, classes, packages and jar files. These units are connected through one of four different relationship types: method call, inheritance, historical and containment. For more information on this data schema, see section IV-C. This call graph is not restricted to a single version of a library but connects methods, classes and packages from all versions of all libraries in the Maven repository. Mathematically, the graph is a collection of tuples connecting two unit identifiers, annotated with one of the four relationship types.

(5) To reduce the size of the Neo4j database, only unique unit identifiers and connections between units are stored. These identifiers are *unitIds* stored in the Berkeley DB database. Neo4j makes fast querying of graph structures possible and also enables the usage of specialized graph queries which relational databases cannot handle (for instance, arbitrarily deep transitive queries). Also, graph traversals can be performed which start at a specific node and visit related nodes to obtain specific information.

(6) The Neo4j graph can be used to query a specific library or a specific method and to investigate changes through time. The graph can also be used to visualize connected units.

(7) The information from Berkeley DB can also be used directly, for instance to obtain a list of all methods present in a certain version of a library or to get information on a specific method in a specific library.

In the next section we discuss limitations and data quality issues present in our dataset.

VI. Limitations

Since the dataset is based on a snapshot of the Maven repository, updates to this repository after the snapshot date are not taken into account into this dataset. Furthermore, users of this dataset should be aware of the following limitations and data quality issues:

A. Skipped libraries

For several reasons, not all libraries have been analyzed:

- Source jars are not available for specific library versions;
- Source jars sometimes contain other languages than Java, contain only test code, property files or binary class files;
- Some source jars are corrupted.

We assume that these missing libraries are randomly distributed over the entire set of libraries, and that they do not introduce a bias in our dataset.

B. Package prefixes

Due to the large size of the dataset it is impossible to manually check data quality. This is also true for package prefixes, which are stored in the *files* table and which were used to calculate isolation ratings as stored in the *deps* table [3]. One problem is that some libraries use multiple package prefixes. For example, `com.thoughtworks.selenium` and `org.openqa.selenium` occur in the same library version. To recognize an import from this library as third-party, both strings have to be recognized. Also, some libraries do not have a common package prefix, making automated detection more difficult. We expect that there does not exist a bias in systems that have missing package prefixes.

C. Usage frequencies

Our dataset also includes usage frequencies of methods, which can be used to determine the expected impact of changes [3]. We calculated these usage frequencies on binary dumps of disassembled class files. This means that the calls present in binary class files can be different from the calls present in source code. This becomes visible, for instance, with calls to `StringBuilder.append`, which is the most frequently called method in the Maven repository. This, however, is caused by the fact that the Java compiler replaces string concatenation using "+" with calls to StringBuilder.

Another issue is whether the usage of libraries by other libraries is representative for the usage of libraries by actual systems. Since a library can be seen as a system in itself we assume that the former is representative for the latter.

D. Wrong snapshot identifiers

A final data problem is the automatic labeling of snapshot numbers as stored in the *snapshotId* column of the *files* table. We used an algorithm from the Maven indexing software itself, but manual inspection shows that subsequent versions sometimes do not get subsequent version numbers. Manual inspection of a sample of jar files shows that data errors like these are only present in a small percentage of files and given the size of the dataset, these errors will not be able to influence large-scale correlations.

VII. Conclusion

We presented MDD, the Maven Dependency Dataset, which contains information on 148,253 Java libraries. We presented conceptual schemas of three different databases. First, we presented a relational database which contains information on individual files and dependencies as well as breaking changes in these files. Next, we presented a key-value database containing information on individual methods, classes and packages. Finally, we presented a graph database which contains all connections between methods, classes and packages of the entire Maven repository. We described our methodology to obtain our data and we discussed data quality issues present in our dataset.

References

[1] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, Washington, DC, USA, 2007. IEEE Computer Society.

[2] S. Raemaekers, A. v. Deursen, and J. Visser. An analysis of dependence on third-party libraries in open source and proprietary system. In *Sixth International Workshop on Software Quality and Maintainability*, mar. 2012.

[3] S. Raemaekers, A. v. Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference on Software Maintenance (ICSM'2012)*, sep. 2012.

Addendum

To the paper *"The Maven repository dataset of metrics, changes, and dependenciesy"*

I. Mentioned websites

The following websites are mentioned in the paper:

Name	*Website*
Apache Maven	http://maven.apache.org
Maven Search	http://search.maven.org
Apache Ant	http://ant.apache.org
DAS-3 Supercomputer	http://www.cs.vu.nl/das3
Clirr	http://clirr.sourceforge.net
Eclipse Evolving API's	http://wiki.eclipse.org/Evolving_Java-based_APIs
Java Language specification on binary incompatibilities	http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html

II. Dataset explanation

A. MySQL database

Tables 1 to 8 show tables and columns present in our MySQL database.

B. Neo4j database

The Neo4j database consists of a collection of tuples of the following form:

```
<unitId1, unitId2, type>
```

where `unitId1` is a 64-bit integer referring to an object in the Berkeley DB database and `unitId2` is a 64-bit integer referring to another object in the Berkeley DB database. These two objects can have one of four types of relationships, which is stored as an integer in `type` and is one of the following:

1) *Next version*
   `unitId1` is the next version of `unitId2`. For instance, when two methods are present in two library versions, one unitId would point to the method in the first version and the other unitId would point to the method in the second version of the library.
2) *Extends/Implements*
   `unitId1` extends or implements `unitId2`. When there is an extends/implements relationship, both identifiers refer to classes or interfaces.
3) *Contains*
   `unitId1` contains `unitId2`. Containment can have different meanings depending on the types of units referred to. For instance, a class can contain a method. If this is the case, `unitId1` refers to a class and `unitId2` refers to a method. Other types of containment are a class that contains another class or a package that contains a class.
4) *Calls*
   `unitId1` calls `unitId2`, which are both methods.

As an example of queries that can be answered using the Neo4j database, consider the following examples:

Count the type of relationships present in the database:

```
START n=node(*)
MATCH n-[r]-m
RETURN type(r), count(*);
```

C. Berkeley DB database

The Berkeley DB database can be used to obtain information on specific methods, classes and packages in the repository. The script `./getunits.sh` can be used to extract information from this database. Below is an example of a query that can be executed:

```
./getunits.sh
  -j <fullmaven.jar path>
  -b <Berkeley DB path>
  -g "tv.bodil"
  -n "tv.bodil.testlol.Testlol.startTimer()"
  -v "1.2.2"
```

The results of this query are as follows:

Property	*Value*
unitId	135108785976600
name	tv.bodil.testlol.Testlol.startTimer()
groupId	tv.bodil
artifactId	maven-testlol-plugin
version	1.2.2
unitType	5 (method)
fileId	12
snapshotId	3
next version	-
LOC	2
McCabe	1
nrParams	0
usageCount	9

To get help using this script, type `./getunits.sh -h`.

References

[1] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, Washington, DC, USA, 2007. IEEE Computer Society.

[2] J. M. Kleinberg. Hubs, authorities, and communities. *ACM Comput. Surv.*, 31(4es), Dec. 1999.

[3] S. Raemaekers, A. v. Deursen, and J. Visser. An analysis of dependence on third-party libraries in open source and proprietary system. In *Sixth International Workshop on Software Quality and Maintainability*, mar. 2012.

files	
Column name	*Description*
fileId	The unique file ID of the file in this table.
fullName	The relative path to the file on disk.
groupId	The groupId of the artifact.
artifactId	The artifactId of the artifact.
version	The version of the artifact.
reservedNodeId	The node at which code was processed on the Supercomputer. Ranges from 1 to 60.
snapshotId	The version number when ordering all library versions from first to latest. Starts with 1.
hasSource	Whether the binary jar has a source jar of the same name in the same directory.
PageRank	(Network metric) The PageRank of the library when representing libraries and their dependencies as a graph.
Betweenness	(Network metric) The betweenness of the library (see [**?**]).
Hubbiness	(Network metric) The hubbiness of the library (HITS-algorithm, see [2]).
Authoritativeness	(Network metric) The authoritativeness of the library (HITS-algorithm, see [2]).
WRM	The Weighted number of Removed Methods compared to the previous version of this library (see [3]).
CEM	The Change in Existing Methods compared to the previous version of this library (see [3]).
RCNO	The Ratio of Change of New and Old methods measured with McCabe differences compared to the previous version of this library (see [3]).
PNM	The Percentage of New Methods compared to the previous verson of this library (see [3]).
nrUnits	The total number of methods in this library version.
nrNewUnits	The number of new methods compared to the previous version of this library.
nrOldUnits	The number of methods that are both in this and the previous version of this library.
nrRemovedUnits	The number of methods that have been removed from the library compared to the last version of this library.
deltaUn	The sum of McCabe values in newly added methods as compared to the previous version.
deltaUo	The difference in McCabe values of existing methods as compared to the previous version.
hws	The weight of this library version as used in summing metric differences over all versions of a library (see [3]).
maintainability	The SIG Maintainability rating of a library (see [1]).
CRS	The Commonality Rating of a System as defined in [4].
RL	The Rating of a Library (indegree, see [4]).
updated	The date that this library version was uploaded to the central repository.
status	Whether the library version has been processed by the Supercomputer.
enabled	Whether the file should be processed. Is false when hasSource = 0 or when there are other reasons this file should be excluded from analysis.
packagePrefix	The "greatest common denominator" of package prefixes as found in the library when scanning for `package` statements. When multiple package prefixes have been found they are separated with a comma. Is used to detect dependencies in other files since these are then imported with `import` statements.

Table 1. Columns in the *files* table, as stored in MySQL.

[4] S. Raemaekers, A. v. Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference on Software Maintenance (ICSM'2012)*, sep. 2012.

deps	
Column name	*Description*
callId	The unique ID of the dependency as stored in this table.
fromFileId	The library that specified the dependency.
toFileId	The library that *fromFileId* depends upon.
isolation	The percentage of files in *fromFileId* that contains an import statement starting with the *packagePrefix* of *toFileId*. For this packagePrefix, see the *files* table.

Table 2. Columns in the *deps* table, as stored in MySQL.

units	
Column name	*Description*
unitId	The unique unit identifier as stored in this table.
name	The fully qualified name of the unit.
groupId	The groupId of the library this unit belongs to.
artifactId	The artifactId of the library this unit belongs to.
version	The version of the library this unit belongs to.
unitType	The unit type of this unit (1 = jar file, 2 = package, 3 = java file, 4 = class, 5 = method)
fileId	The fileId this unit belongs to.
snapshotId	The snapshot number of this unit.
nextVersion	The *unitId* of the next version of this unit.
LOC	The number of lines of source code for this unit.
McCabe	McCabe value for this unit (only when unitType = 5).
nrParams	The number of parameters of this unit (only when unitType = 5)
usageCount	The number of times this unit is being used in the repository (only when unitType = 5)

Table 3. Columns in the *units* table, as stored in Berkeley DB.

changes	
Column name	*Description*
changeId	The unique change ID as stored in this table.
changeType	The type of change as determined by Clirr. For an overview of change types, see *changeTypes*.
fileIdv1	The fileId of the first file involved in the change.
fileIdv2	The fileID of the second file involved in the change.
packageUnitIdv1	the Berkeley DB unitId of the first version of the package involved in the change. Can be null when the change does not involve a package.
packageUnitIdv2	the Berkeley DB unitId of the second version of the package involved in the change. Can be null when the change does not involve a package.
methodUnitIdv1	the Berkeley DB unitId of the first version of the method involved in the change. Can be null when the change does not involve a method.
methodUnitIdv2	the Berkeley DB unitId of the second version of the method involved in the change. Can be null when the change does not involve a method.
classUnitIdv1	the Berkeley DB unitId of the first version of the class involved in the change. Can be null when the change does not involve a class.
classUnitIdv2	the Berkeley DB unitId of the second version of the class involved in the change. Can be null when the change does not involve a class.
fieldUnitIdv1	the Berkeley DB unitId of the first version of the field involved in the change. Can be null when the change does not involve a field.
fieldUnitIdv2	the Berkeley DB unitId of the second version of the field involved in the change. Can be null when the change does not involve a field.

Table 4. Columns in the *changes* table, as stored in MySQL.

changeTypes	
Column name	*Description*
changeTypeId	The unique change type ID as stored in this table.
description	A description of the type of change.
breaking	Whether the change is *breaking*, i.e. whether it causes a *binary incompatibility* in systems using it and which thus have to be recompiled.

Table 5. Columns in the *changeTypes* table, as stored in MySQL.

unitTypes	
Column name	*Description*
unitTypeId	The unique unit type ID as stored in this table.
parentType	The *unitTypeId* of the parent of this *unitTypeId*.
description	A description of the unit type.

Table 6. Columns in the *unitTypes* table, as stored in MySQL.

depTypes	
Column name	*Description*
depTypeId	The unique dependency type ID as stored in this table.
description	A description for this type of dependency.

Table 7. Columns in the *depTypes* table, as stored in MySQL.

stats	
Column name	*Description*
fileId	The fileId to which the statistics belong.
vol	The SIG star rating for volume on a 0.5 - 5.5 scale. 5% of systems has a score between 0.5 and 1.5, 30% has a score between 1.5 and 2.5, 30% has a score between 2.5 and 3.5, 30% has a score between 3.5 and 4.5 and 5% has score between 4.5 and 5.5.
dup	The star rating for duplication on a 0.5 - 5.5 scale.
us	The star rating for unit size (lines of code per method) on a 0.5 - 5.5 scale.
uc	The rating for unit complexity (McCabe) (0.5 - 5.5).
ui	The star rating for unit interfacing (number of parameters per method) (0.5 - 5.5).
mc	The star rating for module coupling (number of incoming dependencies per file) (0.5 - 5.5).
cb	The star rating for component balance (0.5 - 5.5).
ci	The star rating for component independence (0.5 - 5.5).
nm	The number of methods in the system.
nc	The number of classes in the system.
np	The number of packages in the system.
loc	The number of source lines of code in the system.

Table 8. Columns in the *stats* table, as stored in MySQL.

Clirr type	*Description*	*Binary compatible*
1000	Increased visibility of class	Binary compatible
1001	Decreased visibility of class	Breaks compatibility
1002	Unable to determine class scope: in old class version	-
1003	Unable to determine class scope: in new class version	-
2000	Changed from class to interface	Breaks compatibility
2001	Changed from interface to class	Breaks compatibility
3000	Unable to determine whether class is private	-
3001	Removed final modifier	Binary compatible
3002	Added final modifier to class, but class was effectively final anyway	Binary compatible
3003	Added final modifier	Breaks compatibility
3004	Removed abstract modifier	Binary compatible
3005	Added abstract modifier	Breaks compatibility
4000	Added to the set of implemented interfaces	Binary compatible
4001	Removed from the set of implemented interfaces	Breaks compatibility
5000	Added to the list of superclasses	Binary compatible
5001	Removed from the list of superclasses	Breaks compatibility
6000	Added field	Binary compatible
6001	Removed field	Breaks compatibility
6002	Value of field is no longer a compile-time constant	Binary compatible
6003	Value of compile-time constant has been changed	Binary compatible
6004	Changed type of field	Breaks compatibility
6005	Field is now non-final	Binary compatibility
6006	Field is now final	Breaks compatibility
6007	Field is now non-static	Breaks compatibility
6008	Field is now static	Breaks compatibility
6009	Accessibility of field has been increased	Binary compatible
6010	Accessibility of field has been weakened	Breaks compatibility
6011	Field has been removed, but it was previously a constant	Breaks compatibility
7000	Method now implemented in superclass	Binary compatible
7001	Abstract method is now specified by implemented interface	Binary compatible
7002	Method has been removed	Breaks compatibility
7003	Method has been removed, but an inherited definition exists	Binary compatible
7004	Number of arguments changed	Breaks compatibility
7005	Parameter has changed its type	Breaks compatibility
7006	Return type of method has been changed	Breaks compatibility
7007	Method has been deprecated	Binary compatible
7008	Method is no longer deprecated	Binary compatible
7009	Accessibility of method has been decreased	Breaks compatibility
7010	Accessibility of method has been increased	Binary compatible
7011	Method has been added	Binary compatible
7012	Method has been added to an interface	Breaks compatibility
7013	Abstract method has been added	Breaks compatibility
7014	Method is now final	Breaks compatibility
7015	Method is no longer final	Binary compatible
8000	Class added	Binary compatible
8001	Class removed	Breaks compatibility
9000	Unable to determine the accessibility of class	-

Table 9. Detected binary compatibilities and incompatibilities by Clirr.

TUD-SERG-2013-005
ISSN 1872-5392