An Empirical Investigation Into a Large-Scale Java Open Source Code Repository

Mark Grechanik
Accenture Technology Labs
Chicago, IL 60601
drmark@uic.edu

Collin McMillan
The College of William and Mary
Williamsburg, VA 76019
cmc@cs.wm.edu

Luca DeFerrari, Marco Comi, Stefano Crespi
Politecnico di Milano
Milano, Italy
luca.deferrari@mail.polimi.it

Denys Poshyvanyk
The College of William and Mary
denys@cs.wm.edu

Chen Fu, Qing Xie
Accenture Technology Labs
Chicago, IL 60601
chen.fu@accenture.com

Carlo Ghezzi
Politecnico di Milano
Milano, Italy
carlo.ghezzi@mail.polimi.it

ABSTRACT

Getting insight into different aspects of source code artifacts is increasingly important – yet there is little empirical research using large bodies of source code, and subsequently there are not much statistically significant evidence of common patterns and facts of how programmers write source code. We pose 32 research questions, explain rationale behind them, and obtain facts from 2,080 randomly chosen Java applications from Sourceforge. Among these facts we find that most methods have one or zero arguments or they do not return any values, few methods are overridden, most inheritance hierarchies have the depth of one, close to 50% of classes are not explicitly inherited from any classes, and the number of methods is strongly correlated with the number of fields in a class.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Mining software repositories, open source, empirical study, large-scale software, patterns, practice

Keywords

software repository, empirical study

1. INTRODUCTION

A goal of empirical research in software engineering is to establish facts about software from large bodies of experimental data using statistical methods based on probabilistic reasoning [14][20]. Getting insight into different aspects of source code artifacts is increasingly important – it is estimated that around one trillion lines of code have already been written with an additional `35` billion lines of source code being written every year [3]. Yet there is little empirical research using large bodies of source code artifacts, and subsequently there are not much statistically significant evidence of common patterns and facts of how programmers write source code.

One of major requirements for empirical studies is their usefulness [1]. Obtaining statistically significant evidence about commonly used techniques and patterns from the source code may yield important guidance for different areas of computer science. For example, assigning `null` to static fields in Java methods gives an important clue to garbage collectors to remove objects from memory that were bound to these fields. Knowing how commonly programmers assign `null` to static fields may help researchers who work in the area of program analysis to invent heuristic to improve garbage collection [8].

Few major challenges make it difficult to obtain this evidence from source code. First, source code files in large code repositories are treated as unstructured text by search engines and utilities. Since these engines and utilities do not use grammars to parse source code, they do not distinguish between keywords, types, variables, and methods – all program entities are words. Answering even simple questions about programs is very difficult.

Second, source code files are contained in compressed project files which are located in software repositories. Some repositories are well-structures (e.g., FreeBSD) while others are poorly structured (e.g., Sourceforge). Programmers usually submit source code as packaged files of different formats (e.g., archives in bz2 and tar formats). This situation is aggravated by the fact that many application repositories are polluted with poorly functioning projects [10]. Extracting source code files from these projects requires sophisticated fault tolerance mechanisms to handle errors in compressed files and simply not parseable Java code. Finally, users should be able to form declarative queries about different aspects of source code without writing low-level programs that traverse parse trees to collect information in response to these queries. Ideally, queries should be described using SQL, which is a standard declarative language. In our case, it means that source code should be stored in a relational database, and it addresses the first challenge of structuring code.

In this paper we address these major challenges by creating an infrastructure for doing empirical research in source code artifacts. We use this infrastructure to obtain insight into randomly chosen 2,080 Java applications from Sourceforge. We pose 28 research questions, explain rationale behind them when possible, and obtain

```
SELECT COUNT(*) AS totNullAss FROM expression
AS e1, expression AS e2, identifier AS i WHERE
e1.operator='assign' AND e1.op2='null' AND
e1.op1=e2.id AND e2.identifier=i.id AND EXISTS
(SELECT * FROM declaration AS d, decl_modifiers
AS dm WHERE d.id=i.decl AND dm.declaration=d.id
AND dm.modifier='static')
```

Figure 1: An example of SQL statement that obtains information on the number of null assignments to static fields.

different facts from subject applications. Among these facts we find that most methods have one or zero arguments or they do not return any values, few methods are overridden, most inheritance hierarchies have the depth of one, close to 50% of classes are not explicitly inherited from any classes, and the number of methods is strongly correlated with the number of fields in a class.

2. IMPLEMENTATION

In this section we briefly describe how we implemented the infrastructure that we use to obtain answers to research questions.

2.1 Crawlers

The infrastructure consists of two crawlers: *Archiver* that populated the infrastructure's repository by retrieving from Sourceforge more than 30,000 Java projects (some of these projects turned out to be empty) that contain close to 50,000 submitted archive files, which comprise the total of 414,357 files. From these projects we randomly chose 2.080 to study in this paper. *Walker* traverses infrastructure's repository, opens each project by extracting its source code from zipped archive, and applies a parser to the extracted source code. Both crawlers are multithreaded, the Archiver is written in Scala and the Walker is written in Java.

To extract program entities from Java programs we used Sun's class JavaCompiler to parse these programs and build parse trees[1]. Once a parse tree is produced, it is traversed and the content of its nodes is stored in the database. The database contains 71 tables and 278 attributes, and its schema is designed to match (non)terminals of the Java grammar. The database is publicly available[2] for researchers to obtain results for queries that they will state and code using SQL. With this database, our results can be replicated by others researchers.

2.2 Using SQL to State Research Questions

Using SQL enables users to state research questions about Java programs without writing specialized programs that translate these questions into parse tree navigation procedures. Users should know the database schema, understand its relations to the Java grammar, and of course, translate questions from plain English into SQL. For example, the question of how commonly programmers assign `null` to static fields is translated in the SQL statement that is shown in Figure 1.

3. EMPIRICAL EVIDENCE

In this section we state research questions (RQs) and describe the results that we obtain to answer these questions. Table 1 lists RQs that address how programmers use Java language features in their programs. The numbers next to paragraphs below corresponds to the numbers of rows in the Table 1.

[1]http://java.sun.com/javase/6/docs/api/javax/tools/JavaCompiler.html
[2]http://www.cs.wm.edu/semeru/treasure

3.1 Classes and Packages

1. We collected information about 270,973 Java classes that are contained in 2,080 Java applications. Out of these classes, 5,827 or a little over 2% are declared as abstract. On average, a Java application contains close to 97 classes (median 33) with the range from 1 to 2017 classes. The histogram of the distribution of classes per applications is shown in Figure 2(a). The horizontal axis shows the range of classes for each bar, whose height shows the number of applications that contain this range of classes.

2. There are 7,368 static classes, less than 3% of the total number of classes. In a population of Java applications that contain static classes, an application contains close to 7 static classes on average (median is equal to zero) with the range from 0 to 1,035 static classes. The histogram of the distribution of static classes per applications is shown in Figure 2(b).

3. There are 29,237 anonymous classes, less than 11% of the total number of classes. The histogram of the distribution of anonymous classes per container class is shown in Figure 2(c) using the log scale.

4. There are 14,270 nested classes, less than 6% of the total number of classes. The histogram of the distribution of nested classes per container class is shown in Figure 2(d). The horizontal axis shows the range of nested classes for each bar, whose height (using the log scale) shows the number of classes that contain this range of classes.

5. Inheritance is a key mechanism of object-oriented programming to achieve reuse [5]. To determine how programmers use inheritance, we computed all inheritance hierarchies of the depth one or more. Implicitly inheriting from the class `java.lang.Object` was not counted. Out of a total of 11,298 hierarchies, 8,008 or almost 71% have the depth one. The maximum inheritance depth that we found is five. The histogram of the distribution of inheritance trees per inheritance depth is shown in Figure 2(e). The horizontal axis shows the depth of inheritance for each bar, whose height shows the number of independent inheritance trees.

For example, one application that contains the maximum level of inheritance hierarchy is LOCKSS[3], it is developed by Stanford University Libraries. It contains 963 Java classes with the total of 228,076 lines of code. One inheritance hierarchy from this application is `AdminIpAccess`→`IpAccessControl`→`LockssServlet`→`HttpServlet`→`GenericServelet`→`Object`. The package `org.lockss.uiapi.commands` contains these classes. The semantics of this inheritance hierarchy has something to do with implementing a command protocol.

There are 115,453 classes that do not inherit explicitly from any class versus 116,194 classes that participate in some inheritance hierarchies. That is, almost 50% of classes are written without using inheritance. There are 9,467 classes or 4% that are the roots on inheritance trees. The histogram of the distribution of classes per inheritance depth is shown in Figure 2(f) using the log scale. The horizontal axis shows the depth of inheritance for each bar, whose height shows the number of classes that belong to the inheritance trees of the given depth.

6. How classes are distributed among inheritance trees is shown in Figure 2(g) using the log scale. Majority of inheritance trees (i.e., more than 80%) contain from two to five classes. Less than 9% of trees contain more than ten classes. The median is three classes per inheritance hierarchy.

7. Interface implementation is a key mechanism for subtyping in object-oriented programming [5]. To determine how programmers use interface implementations, we computed all hierarchies of the

[3]http://www.lockss.org

depth one or more. Out of a total of 2,026 interface extend hierarchies, 1,664 or more than 82% have the depth one or more. The maximum depth that we found is five. The histogram of the distribution of interface extend trees per depth is shown in Figure 2(h). The horizontal axis shows the depth of interface implementation hierarchy for each bar, whose height shows the number of independent interface trees.

8. There are 17,254 methods that are overridden in derived classes, or less than 2% of the total of 938,779 found methods. We found that in parent/child class relationships there is 1.6 methods overridden on average with the median equal to one. Method overriding facilitates polymorphism and this result shows that this overriding feature is used by programmers but not actively. The histogram of the distribution of method overriding cases by the number of overridden methods is shown in Figure 2(i) using the log scale. The horizontal axis shows the ranges of numbers of overridden methods for each bar, whose height shows the number of inheritance trees where the number of the overridden methods is present.

9. There are 2,047 assert statements into Java applications. On average, there is one assert statement per 400 methods (median zero). The histogram of the distribution of imports by the number of applications is shown in Figure 2(j) using the log scale.

3.2 Constructors and Methods

10. We found 938,779 methods that are defined in 270,973 Java classes. On average, there are 3.5 methods per class (median equal to four) with the range from 1 to 1175 methods per class. The histogram of the distribution of methods per classes is shown in Figure 3(a).

There are 145,124 classes that do not define constructors, which is over 53% of the total number of classes. Almost 40% of classes have overloaded constructors. The histogram of the distribution of the classes by the number of overloaded constructors is shown in Figure 3(b) using the log scale.

11. In Java, programmers can call methods recursively, and we found 35,846 occurrences of recursive method calls in the definitions of 938,779 methods (less than 4%). On average, there is one recursive call per 23 methods, median is zero. We count each functional call that may result in cycles in the call graph as one recursive call. The histogram of the distribution of the recursive method calls by the number of methods is shown in Figure 3(c) using the log scale.

12. Close to 25% of methods are static (excluding the method `main`). On average, there are 0.85 static methods per class (median equal to zero) with the range from 1 to 289 static methods per class. The histogram of the distribution of methods per classes is shown in Figure 3(d).

13. We found 84,130 methods that are declared in 24,875 Java interfaces. On average, there are 3.4 methods per interface (median equal to three) with the range from 1 to 558 methods per interface. The histogram of the distribution of methods per interfaces is shown in Figure 3(e).

14. The arity of a method is the number of arguments that this method takes. We found that over 42% of methods do not take any arguments, that is their arity is zero. The remaining 544,324 methods take from one to the maximum of 30 arguments, with the average of 1.5 arguments per method (median is equal to one). The histogram of the distribution of arities per methods is shown in Figure 3(f).

15 and 16. Methods that do not return values (i.e., return type `void`) constitute over 44% of all methods. On average, classes that contain methods that do not return values have 5.1 methods versus 5.8 methods that return values of some type. A total of 91,423 methods take no arguments and do not return any values. **Based on 14, 15, and 16 we observe that programmers prefer to define methods that take one or zero arguments or do not return any values.**

17. A total of 24,744 methods or less than 3% return arrays. Counting only classes that contain these methods, we found that there are two methods per class on average (median is one) with maximum of 137 methods per class.

18. The keyword `this` is used in 840,937 methods and constructors to invoke different methods and constructors 3,979,285 times (i.e., `this.methodname(..)`). That is, the keyword `this` is used in almost 90% of methods. Counting only methods that contain `this`, we found that there are 2.2 uses of `this` per method on average with the maximum of 785 uses per method. Third edition of Java says that using `this` "... is simpler than having to invent different names for the parameters and is not too confusing in this stylized context[4]. In general, however, it is considered poor style to have local variables with the same names as fields."

3.3 Fields

In this section we provide evidence of how programmers use constructors and methods when they write programs.

Methods and Fields. The graph for correlations the numbers of methods with the numbers of fields in classes is shown in Figure 4(d). Correlation coefficient is equal to 0.99 that enables us to conclude that **the number of fields in a class is strongly correlated with the number of methods in the same class**. This result may have different implications for the tasks of maintenance and evolution of applications, for example, to predict different metrics by knowing how many fields a class holds.

19. We collected information about 448,898 fields that are declared in 270,973 Java classes. On average, a Java class contains close to two fields (median zero) with the range from zero to 1,457 fields. The histogram of the distribution of classes per applications is shown in Figure 3(g).

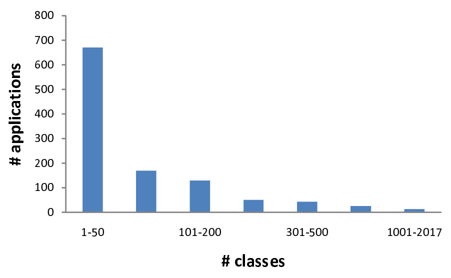
20. In Java, threads can access shared variables and keep private working copies of the variables to achieve a more efficient implementation of multiple threads. Java offers a mechanism to programmers, called volatile fields to ensure that working copies of variables should be reconciled with the master copies in the shared main memory when objects are locked or unlocked. We found a total of 492 volatile fields out of a total of 448,898 fields. On average, one in 500 Java classes contains volatile fields (median zero) with the range from zero to nine volatile fields. The histogram of the distribution of volatile fields per classes is shown in Figure 3(h).

Transient fields are not saved as part of the persistent states of Java objects. We detected a total of 2,305 transient fields; the histogram of the distribution of transient fields per classes is shown in Figure 3(i).

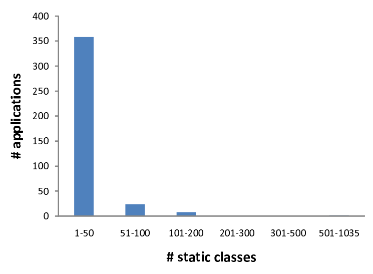
21. Out of 448,898 fields, 154,067 or 34% are declared static. On average, a Java class contains close to 0.7 static fields (median zero) with the range from zero to 1,457 fields. The histogram of the distribution of static classes per applications is shown in Figure 3(j).

Assignments of `null` to static fields signal garbage collector that objects that are used to be assigned to these fields, can be collected from memory [8]. Knowing how often programmers assign null to static fields can serve as a rule-of-thumb for researchers to develop tools and techniques that use this heuristics to improve memory management.

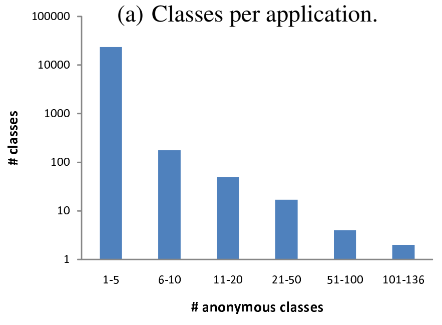
[4]http://java.sun.com/docs/books/jls/third_edition/html/statements.html



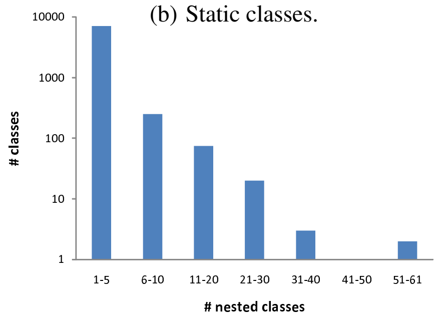
(a) Classes per application.



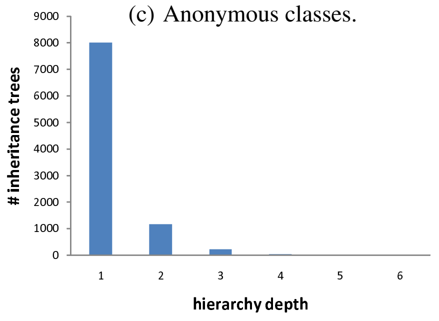
(b) Static classes.



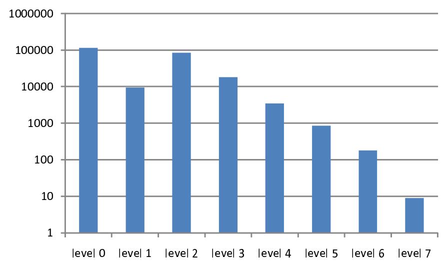
(c) Anonymous classes.



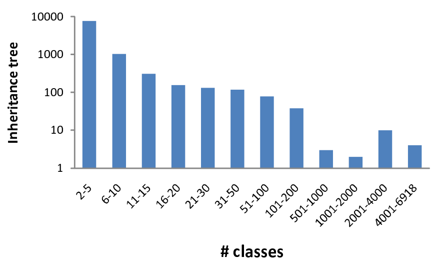
(d) Nested classes.



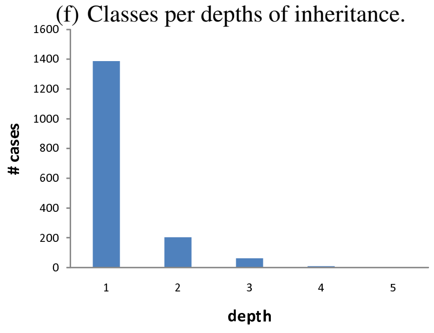
(e) Inheritance hierarchies.



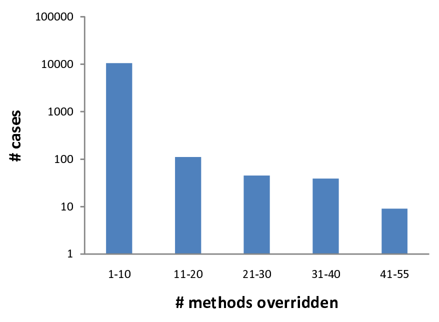
(f) Classes per depths of inheritance.



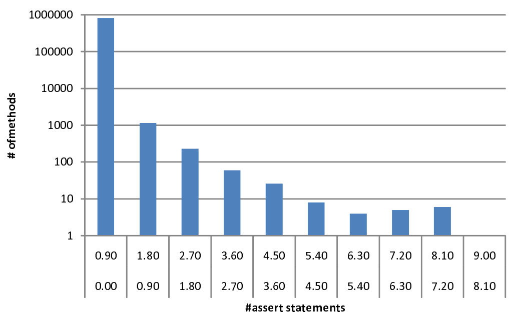
(g) Classes per inheritance trees.



(h) Interface extends per depth.

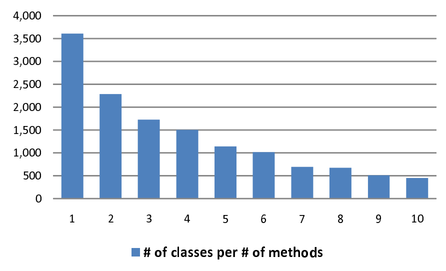


(i) Methods overridden per inheritance case.

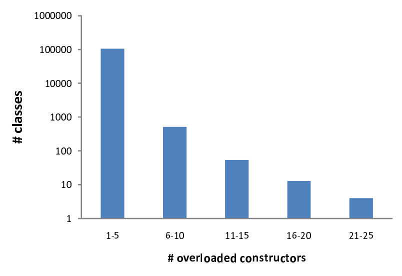


(j) Assert statements per methods.

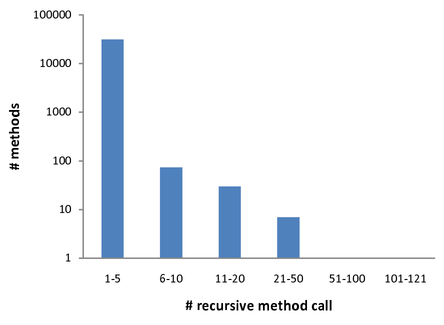
Figure 2: Histograms of classes, inheritance, method overriding, and assert statements.



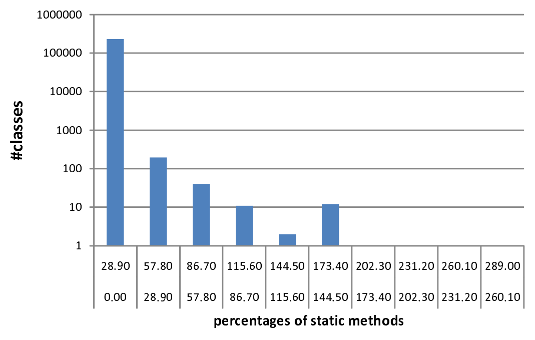
(a) Methods per classes.



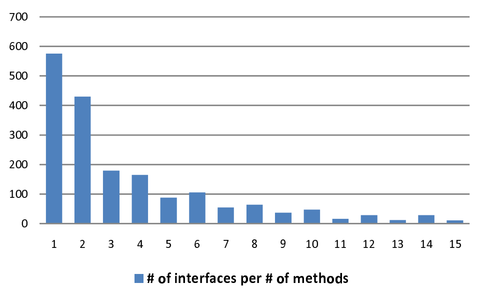
(b) Overloaded constructors per classes.



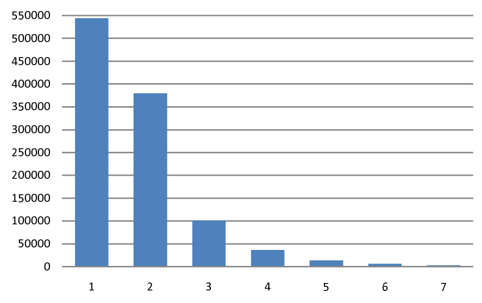
(c) Recursive method calls.



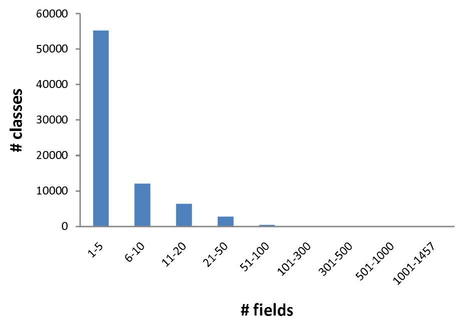
(d) Static methods per classes.



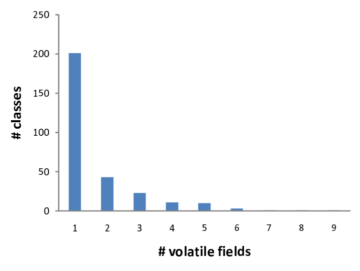
(e) Methods declared in interfaces.



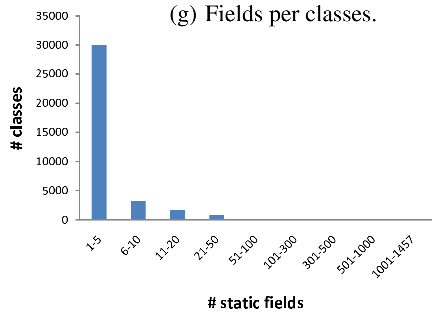
(f) Arities of methods.



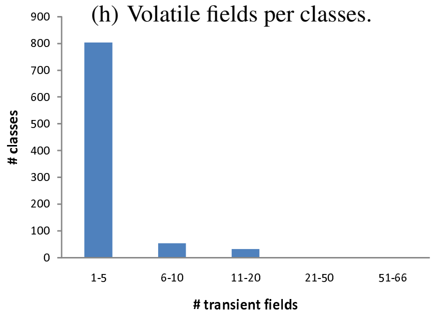
(g) Fields per classes.



(h) Volatile fields per classes.



(i) Transient fields per classes.



(j) Static fields per classes.

Figure 3: Histograms for methods, constructors, and fields.

We found a total of 831 assignments of `null` to static fields, out of a total of 29,907 assignments to static fields. In less than 2.8% of these assignments programmers used the value `null`.

22. Out of 448,898 fields, 231,647 or almost 52% have the type `String`. Approximately one in three classes has between one to 432 fields of the type `String`, making it one of the most popular types.

3.4 Statements

23. We found 620,419 conditional statements (i.e., `if-else`, `switch`, `for`, `while`, `do-while`) that are used in 938,779 methods, out of which 397,605 methods or approximately 64% do not have any conditional statements. Approximately 42% of the `switch` statements do not contain the `default` path. On average, there are approximately 0.8 statements per method (median equal to zero) with the range from 0 to 750 statements per method. The histogram of the distribution of these statements per methods is shown in Figure 4(a).

24. Out of 620,419 conditional statements only 4,956 (less than 1%) use simple boolean variables as their conditions. That it, most statements use expressions, and thus analyzing code to determine execution paths requires sophisticated reasoning. The histogram of the distribution of these statements per methods is shown in Figure 4(b).

Return versus Break. The graph for usages of the keywords return versus break within conditional statements is shown in Figure 4(c). The keyword `return` is most popular with the `for` statements – we counted 2,333,777 occurrences and the least popular with the `switch` statements – only 46,044 occurrences. Conversely, the keyword `break` is most popular with the `for` statements – we counted 245,502 (the statement `switch` comes next with 102,454 occurrences) occurrences and the least popular with the `do-while` statements – only 16,975 occurrences. In general, the keyword `return` is used much more often than `break` for all types of conditional statements except for `switch`, where break is used more than twice as often.

3.5 Exceptions

25. We found 93,714 `try/catch` statements, on average one statement per ten methods (median equal to zero) with the range from 0 to 90 statements per method. The block `finally` is used in 6.8% after `try/catch` statements. The histogram of the distribution of these statements per methods is shown in Figure 4(e).

26. We found 19,181 exceptions that are thrown using the keyword `throw`. Considering only methods that throw exceptions, on average, close to one method throws an exception (median equal to zero) with the range from 0 to 40 exceptions thrown per method. The histogram of the distribution of these statements per methods is shown in Figure 4(f).

27. Exception propagation is a standard technique for throwing exceptions from `catch` blocks to propagate these exceptions to desired points at execution where they will be handled. We found 110,740 propagated exceptions. Approximated 14% of exceptions are propagated with 0.26 propagated exceptions per `catch` block on average (median equal to zero) with the range from 0 to 5 exceptions thrown per `catch` block. The histogram of the distribution of propagations per `catch` block is shown in Figure 4(g).

3.6 Variables and Basic Types

28. We collected information about 5,775,367 local variables that are declared in method scopes. On average, a method contains close to three local variables (median zero) with the range from zero to 1,055 variables. Out of these variable, approximately 10% are declared `final`. The histogram of the distribution of variables per methods is shown in Figure 4(h) using the log scale.

Using basic types versus classes that represent these types. It may come as no surprise that we found that programmers prefer to use basic types (e.g., `int`, `boolean`) to their corresponding class-based types (e.g., Integer, Boolean). However, we find it surprising that the difference is close to two orders of magnitude in favor of basic types. The most used basic type is `int` followed by `boolean` followed by `long` followed by `double`, `float`, `byte`, and `short` in this order. The order for class-based types is a bit different – Long and Boolean switch their respective positions. The histograms of the distribution of class and basic types are shown in Figure 4(i) and Figure 4(j) respectively.

Using increment ++ and decrement - - operators. We found 25,523 uses of the increment operator versus only 2,005 uses of decrements. That is, the number of increments is greater by more than the order of magnitude than the number of decrements.

3.7 Evolution and Maintenance

Evolution and maintenance of software account for over 70% of the total cost of software projects [7, page 23]. Knowing how programmers modify code is important to justify and develop new approaches and techniques that help these programmers to deal with evolution and maintenance effectively. For example, step-wise refinement is a widely accepted approach for developing a complex program from a simple program by incrementally adding details [6]. Yet the question is if programmers follow this approach at large. We provide partial answers to this question in this section.

29. We downloaded multiple versions of applications from Sourceforge, a total of 2,427 or on average 1.5 versions per application with the median of one, that is two releases of the same application. Zero versions means only one release, and we located a maximum of 24 releases of a single application. The histogram of the distribution of the numbers of applications per versions is shown in Figure 5(a).

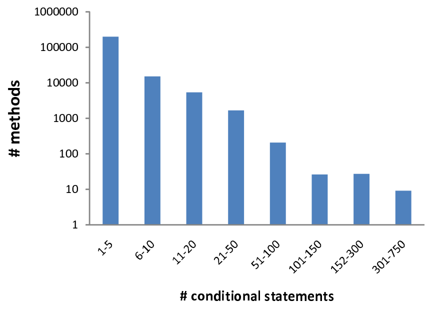
30. We computed the numbers of added and removed fields between different versions of the same applications. All fields whose names or types were changed were counted as added/removed. There are on average 2.57 fields added between versions of the same application with the median of 2. A maximum of 286 fields are added with a maximum of 190 fields removed. The histogram of the distribution of the numbers of added/removed fields per versions is shown in Figure 5(b).

31. We computed the numbers of added and removed methods between different versions of the same applications. All methods whose signatures were changed were counted as added/removed. There are on average 3.6 methods added between versions of the same application with the median of 3. A maximum of 262 methods are added with a maximum of 436 fields removed. The histogram of the distribution of the numbers of added/removed methods per versions is shown in Figure 5(c).

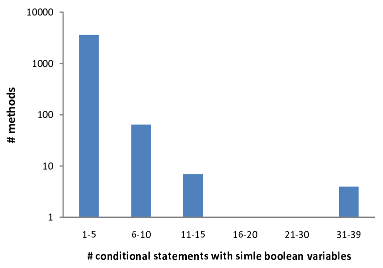
32. We computed the numbers of added and removed classes between different versions of the same applications. All classes whose names were changed were counted as added/removed. There are on average 96.3 classes removed between versions of the same application with the median of four classes removed. A maximum of 1,090 classes are added with a maximum of 1,845 classes removed. The histogram of the distribution of the numbers of added/removed classes per versions is shown in Figure 5(d).

4. RELATED WORK

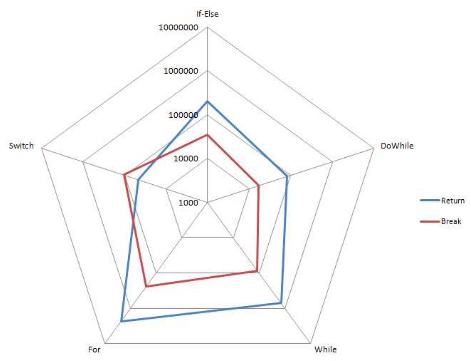
The work presented in this paper falls into the following two major categories: (1) Infrastructure, tools and techniques that fa-



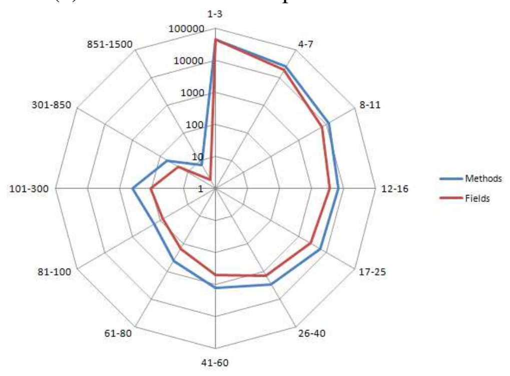
(a) Conditional statements per method.



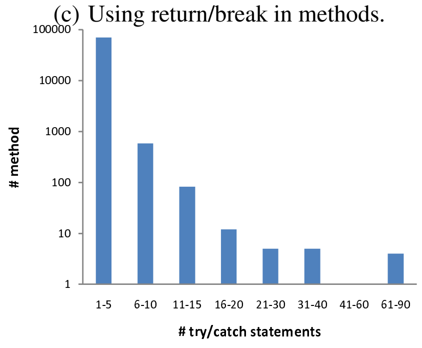
(b) Statements with simple boolean vars.



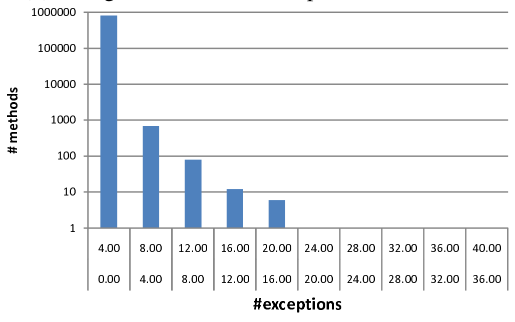
(c) Using return/break in methods.



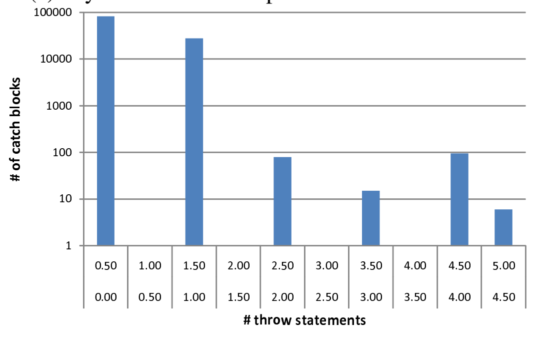
(d) Using methods and fields per classes.



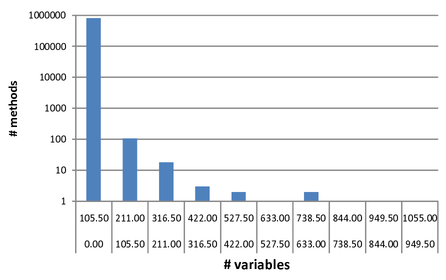
(e) Try/catch statements per method.



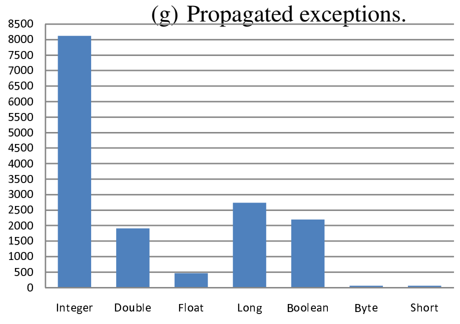
(f) Exceptions per methods.



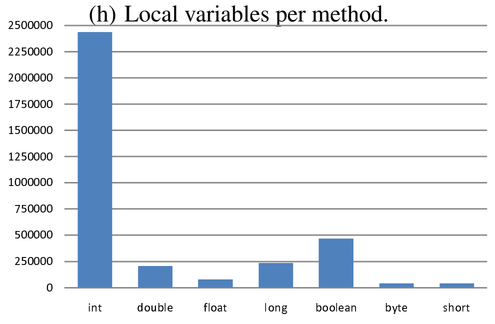
(g) Propagated exceptions.



(h) Local variables per method.



(i) Classes that represent basic types.



(j) Basic types per classes.

Figure 4: Histograms for conditional statements, exceptions, variables, and types.

#	Research Question	Total	Mean	Med	Max	Min	σ^2	Var	Kur	Ske	CI
1	Classes, per application	270,973	96.8	33	2071	1	193.2	37,320	30.5	4.8	11.4
2	Static classes	7,368	6.7	0	1,035	0	38.5	1484	488	19.7	2.3
3	Anonymous classes	29,237	0.036	0	136	0	0.4	0.16	38595	146	0.001
4	Nested classes	14,270	0.06	0	61	0	0.56	0.3	1819	30.1	0.002
5	Inheritance hierarchies (no zero depth)	11,298	1.2	1	7	1	0.5	0.26	12.9	3.2	0.01
6	Classes per inheritance tree	116,194	12.3	3	6,918	2	155	24,116	909	28.4	0.99
7	Interface implementation trees	2026	1.2	1	5	1	0.54	0.3	8.7	2.8	0.03
8	Methods of parent classes that are overridden by derived classes	17,254	1.6	1	55	1	3.38	11.4	77.3	8.2	0.06
9	Assert statements per methods	2,047	0.0025	0	9	0	0.07	0.005	3426	47	0.002
10	Methods per classes	938,779	3.5	4	1175	1	28.6	818	415	16.8	13.8
11	Recursive method calls	35,846	0.04	0	121	0	0.31	0.1	33830	108	0.001
12	Static methods per class (except `main`)	231,647	0.36	0	289	0	2.7	7.5	1586	28.4	0.35
13	Methods per interfaces	84,130	3.4	3	558	1	15.1	229	345	14.8	0.26
14	Arities of methods	544,324	1.5	1	30	1	1.06	1.1	39.5	4.2	1.5
15	Methods that return `void`	414,953	5.1	3	1172	1	13.4	178	655	45	5.1
16	Methods that return non-`void`	523,826	5.8	3	888	1	14.1	199	655	20	5.7
17	Methods that return arrays	24,744	2	1	137	1	2.7	7.3	761	18.8	2
18	Using `this`, per method	840,937	2.2	1	785	0	5.6	31.6	3983	48	2.2
19	Class fields	448,898	1.9	0	1,457	0	8.4	71	5923	51	0.03
20	Volatile fields	492	0.002	0	9	0	0.075	0.006	3892	54.4	0.0003
21	Static fields	154,067	0.7	0	1457	0	6.3	39.3	15807	93	0.03
22	Fields of type `String`, per class	231,647	0.3	0	432	0	2.5	6.2	7738	65.3	0.3
23	Conditional statements, per method	620,419	0.76	0	750	0	3.3	11	10417	65.7	0.007
24	Conditional statements that use simple boolean variables as their conditions	4,956	0.006	0	39	0	0.14	0.02	25482	114	0.0003
25	`Try/catch` statements, per method	93,714	0.11	0	90	0	0.51	0.26	4571	34.7	0.001
26	Exceptions, per method	818,358	0.9	0	40	0	0.26	0.07	2059	29.2	0.02
27	Exceptions thrown from catch blocks	110,740	0.26	0	5	0	0.45	0.2	3.9	1.6	0.26
28	Local variable per method	818,358	0.87	0	1055	0	4.6	20.8	6023	43	1.35
29	Versions per applications	2,427	1.5	1	24	0	1.86	3.45	58.3	6.8	1.38
30	Removed/added fields between versions	6,249	2.57	2	286	-190	14	194.7	54.8	0.2	2.2
31	Removed/added methods between versions	7,861	3.6	3	262	-438	21.5	464.3	111.4	-5.5	3.2
32	Removed/added classes between versions	5,713	-96.3	-4	1,090	-1,845	379.5	144084	11.2	-2.54	-181

Table 1: Research question and answers.

cilitate learning source code artifacts, and (2) empirical studies and analyses that review and explain the correlations of these artifacts.

Infrastructure. Howison et al. [9] created a collaborative data and analysis repository, called FLOSSMole in order to gather, share, and store comparable data and analyses of open-source projects. Our paper is different from this work, since we collect and analyze the data at the source code level of OSS projects in large repositories, whereas FLOSSMole gathers metadata (e.g., project topics, activity, statistics, licenses, developer skills etc).

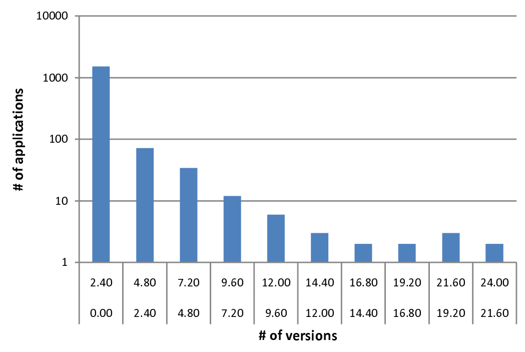
Kiefer et al. [12] proposed an EvoOnt, a software repository data exchange format based on a Web ontology language to facilitate mining software repositories for software analysis. Contrary to EvoOnt, we ask research questions at source-code level and answering those RQs with EvoOnt is not currently possible since it will require a significant extension of EvoOnt.

Mockus describes some tools and techniques for gathering, indexing, and updating of a large universal repository for publicly accessible version control systems [16]. While the work of Mockus concentrates on creating a Universal Version History of publicly available software projects, we answer specific research questions using statistical methods.

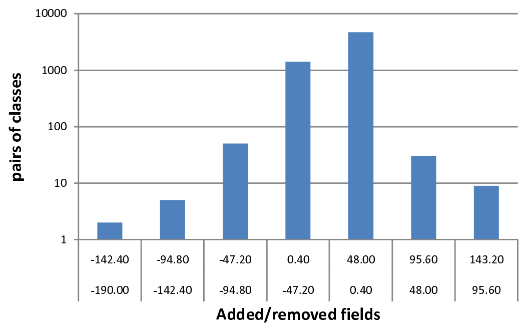
In a different paper, Mockus et al. examined data from two major open source projects, Apache and Mozilla, to investigate whether open source style of software development can compete with traditional commercial development method successfully in terms of developer participation, core team size, code ownership, productivity, defect density, and problem resolution intervals [17]. This study is more focused on the artifact of change and maintenance of the software than statistics about source code.

Ossher et al. built SourcererDB, an aggregated repository of 2,852 statically analyzed and cross-linked open-source Java projects from sourceforge, Apache and Java.net [18]. All the extracted data in SourcererDB is stored in a relational database, similarly to our approach. Our work is different from SourcererDB, since we address and statistically analyze this data to provide important insights into open-source software development practices.

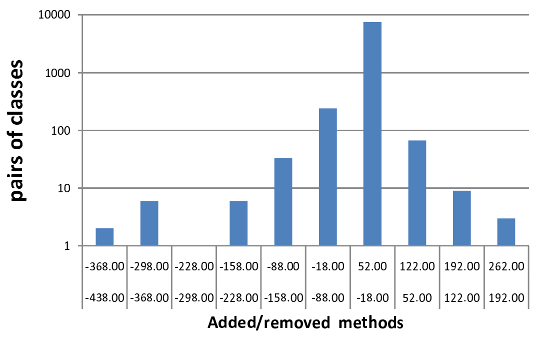
Studies and analyses. Baxter et al. presented the first-in-depth study on the structure of Java programs [2]. In their study, they analyzed 56 Java applications and measured several key structural attributes (e.g., number of methods, number of fields etc.) to see if these structural characteristics obey power laws. While our study investigates similar structural characteristics to those studied in Baxter et al., we explore more different research questions, we perform our study on a significantly larger dataset (that is several



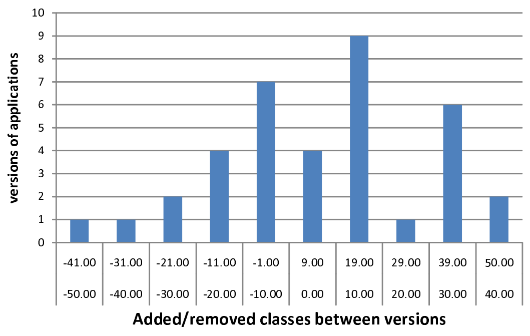
(a) Versions per applications.



(b) Removed/added fields across versions of applications.



(c) Removed/added methods across versions of applications.



(d) Removed/added classes across versions of applications.

Figure 5: Histograms for evolution of applications.

thousand programs) and we focus on practical applications of our findings, which can be addressed while gleaning this information from large repositories.

Kim and Whitehead analyzed signature change properties of seven software project histories and revealed multiple properties of signature changes [13]. While in our current work we do not address questions, which are relevant to the evolution of signature change patterns, we are planning to tackle these research questions on our dataset for the future work.

Jiau and Kao [11] examined correlation between user and developer activities and the state of development of open-source projects. This study is structurally similar to our paper, however, their research questions are not concerned with source code level details like ours.

Koch [15] explores possible benefits of communication and coordination tools in Sourceforge open-source projects. Our approach, on the other hand, obtains information from the source code artifacts.

The work by Succi et al. [21] is relevant to our work as it statistically explores various properties of source code elements (that is, computing metrics from Chidamber and Kemerer suite [4]) on a large data set of publicly available projects. While the work by Succi et al. focuses on analyzing correlations among software metrics extracted from a number of open-source projects, our work is concerned with gaining insights into a large number of research questions.

Finally, the work by Shang [19] can be used to speed up and scale up analysis and mining software repositories using distributed computing platforms. We plan to rely on MapReduce in our future work to reduce computation time for updating and analyzing projects in our infrastructure.

5. CONCLUSION

In this paper we describe an infrastructure for doing empirical research in source code artifacts. Using this infrastructure we obtain insight into over 2,080 Java applications. We pose 32 research questions, explain rationale behind them, and obtain results that shed light into the practice-at-large of writing Java code.

Acknowledgments

We warmly thank anonymous reviewers for their comments and suggestions that helped us to improve the quality of this paper. This work is supported by NSF CCF-0916139, CCF-0916260, Accenture, and United States AFOSR grant number FA9550-07-1-0030. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

6. REFERENCES

[1] J. Armstrong. Discovery and communication of important marketing findings: Evidence and proposals. General Economics and Teaching 0412011, EconWPA, Dec. 2004.

[2] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of java software. *SIGPLAN Not.*, 41(10):397–412, 2006.

[3] G. Booch. Keynote speech: The complexity of programming models. In *AOSD*, page 1, 2005.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[5] W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[7] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[8] B. Goetz. Java theory and practice: Garbage collection and performance. *IBM Developers Work*, 2004.

[9] J. 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, 07/2006 2006.

[10] J. Howison and K. Crowston. The perils and pitfalls of mining Sourceforge. In *MSR*, 2004.

[11] H. C. Jiau and C. H. Kao. Assessing the efficacy of user and developer activities in facilitating the development of oss projects. *J. Softw. Maint. Evol.*, 21(5):287–314, 2009.

[12] C. Kiefer, A. Bernstein, and J. Tappolet. Mining software repositories with isparol and a software evolution ontology. In *MSR '07*, page 10, Washington, DC, USA, 2007. IEEE Computer Society.

[13] S. Kim, E. J. Whitehead, and . J. Bevan, Jr. Properties of signature change patterns. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 4–13, Washington, DC, USA, 2006. IEEE Computer Society.

[14] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, 2002.

[15] S. Koch. Exploring the effects of sourceforge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empirical Softw. Engg.*, 14(4):397–417, 2009.

[16] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *MSR '09*, pages 11–20, Washington, DC, USA, 2009. IEEE Computer Society.

[17] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[18] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In *MSR '09*, pages 183–186, Washington, DC, USA, 2009. IEEE Computer Society.

[19] W. Shang, Z. M. Jiang, B. Adams, and A. E. Hassan. Mapreduce as a general framework to support research in mining software repositories (msr). In *MSR '09*, pages 21–30, Washington, DC, USA, 2009. IEEE Computer Society.

[20] D. I. K. Sjoberg, T. Dyba, and M. Jorgensen. The future of empirical methods in software engineering research. In *FOSE '07: 2007 Future of Software Engineering*, pages 358–378, Washington, DC, USA, 2007. IEEE Computer Society.

[21] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo. An empirical exploration of the distributions of the chidamber and kemerer object-oriented metrics suite. *Empirical Softw. Engg.*, 10(1):81–104, 2005.