String Representations of Java Objects: An Empirical Study

Matúš Sulír[0000−0003−2221−9225]

Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
matus.sulir@tuke.sk

Abstract. String representations of objects are used for many purposes during software development, including debugging and logging. In Java, each class can define its own string representation by overriding the toString method. Despite their usefulness, these methods have been neglected by researchers so far. In this paper, we describe an empirical study of toString methods performed on a corpus of Java files. We are asking what portion of classes defines toString, how are these methods called, and what do they look like. We found that the majority of classes do not override the default (not very useful) implementation. A large portion of the toString method calls is implicit (using a concatenation operator). The calls to toString are used for nested string representation building, exception handling, in introspection libraries, for type conversion, and in test code. A typical toString implementation consists of literals, field reads, and string concatenation. Around one third of the string representation definitions is schematic. Half of such schematic implementations do not include all member variables in the printout. This fact motivates the future research direction – fully automated generation of succinct toString methods.

Keywords: ToString methods · Java · Quantitative study · Qualitative study.

1 Introduction

Almost every object-oriented programming language supports a way to represent the state of an object as a text string. Usually, the textual representation is obtained by calling a method such as `String()`, `printOn:`, or `to_s`.

In Java, this method is called `toString`. Its basic implementation is defined in the root of the class hierarchy – the class `Object`. It is therefore callable on an object of any type. However, the default implementation displays only a class name and a hash code of the object, e.g., `java.io.FileWriter@5c29bfd`. Each class can override this method to implement a potentially useful textual representation. For example, a `HashMap` instance can be represented as "{a=1, b=2}".

String representations are used during debugging, logging, testing and for many other purposes. For instance, when a developer displays a list of variables

This is a post-peer-review, pre-copyedit version of an article published at SOFSEM 2020. The final authenticated version is available online at https://doi.org/10.1007/978-3-030-38919-2_39.

in a contemporary debugger, each non-primitive variable is initially shown as a string obtained by calling the `toString` method on it. Only after expanding it, individual member variables are displayed.

Despite their importance, `toString` and related methods have received limited attention by researchers so far. Schwarz [10] performed a very brief analysis of `printOn:` methods in Smalltalk. Although not directly focused on string representation methods, Qiu et al. [8] and Lemay [5] performed large-scale studies of Java language usage, where they found the `toString` method is among the most used standard library calls.

In this paper, we would like to present an empirical study about the prevalence and properties of `toString` methods in Java. Using automated analysis of a source code corpus and manual inspection of selected examples, we will answer the following research questions:

- **RQ1**: What portion of classes defines their own `toString` method?
  - **RQ1.1**: How deep in the inheritance hierarchy are `toString` methods defined?
- **RQ2**: How are `toString` methods called in the code?
  - **RQ2.1**: Are they explicit (method calls) or implicit (concatenations of a string and a non-string)?
  - **RQ2.2**: Are they often called from other `toString` implementations?
  - **RQ2.3**: What are other common usage scenarios of string representations?
- **RQ3**: What do typical `toString` methods look like?
  - **RQ3.1**: What language constructs do they consist of?
  - **RQ3.2**: Do they often call `toString` of the superclass?
  - **RQ3.3**: Are they rather schematic, or do they contain advanced logic?
  - **RQ3.4**: Do they print the values of all member variables or only a portion of them?

The contribution of this paper is twofold. First, we expand the available knowledge in this area by describing the current state of string representation implementation and usage. Second, since these implementations may be repetitive, this paper can act as a basis for their future automated generation. This could, for example, make debugging easier, particularly in situations when the available display space is constrained [12].

In section 2, we briefly describe the method. In sections 3, 4, and 5, the method is elaborated and the results of each research question are presented. Finally, we describe the threats to validity (section 6), related work (7), conclusion and future work (8).

2 Method Outline

To perform the analysis, we needed a corpus of parsable source code files. For some questions or their parts, type resolution was necessary. This means that except for the source files themselves, we needed also their compiled versions

along with all their dependencies, such as third-party libraries. Furthermore, although projects in online repositories (e.g., GitHub) are easily accessible in large quantities, they can include homework assignments or personal backups, potentially skewing the analysis results [4]. Therefore, we sought for a curated collection of software projects.

The only well-known corpora fulfilling all of the above criteria are the Qualitas Corpus [13] and corpora based on it, such as Qualitas.class [14]. Since they are outdated, we decided to build a new corpus based on them. From the Qualitas Corpus [13], we selected a random subset of 15 open source projects which are still active (updated in the last year) and successfully buildable. We manually downloaded their current versions and whenever possible, we also automatically downloaded both the source code and binary forms of their dependencies, including the transitive ones. This way, we obtained an up-to-date corpus consisting of 759 artifacts (projects and modules) with 106,473 unique Java files (based on the package and type names). The number of classes in these files, including named inner and nested classes, is 149,057. For more than 97% of them, we were able to successfully resolve all their superclasses up to the root of the type hierarchy.

After constructing the corpus, we executed an automated source code analysis process on it. This analysis was performed by a custom program we have written using the Spoon library [7]. To enable the reproducibility of research, both the corpus-building script and the analysis program are available online[1].

All quantitative results were produced fully automatically by the program. The output of the analysis included also source code examples, some of which we inspected manually, thus producing qualitative results (RQ2.3 and a part of RQ3.1). Details of the method are different for each of the research questions, so they are described in the following sections.

3 ToString Definitions

First of all, we would like to know what proportion of classes defines their own `toString` method and how many of them rely on the default representation. A naive approach answering this might be to count the classes defining `toString` and divide them by the total number of classes. However, we must consider also type hierarchies: For example, a `HashMap` does not define its own `toString`, but its superclass, `AbstractMap`, defines one. A string representation defined in a superclass is usually still useful, but in some cases it may be less specific than the method defined directly in the class.

Therefore, we first counted all concrete (not abstract) classes, excluding enumeration types, anonymous and test classes, considering only classes with resolved supertypes. The result was a list of 94,548 classes. Then for each of them, we constructed a type hierarchy from the class itself up to `Object`. Finally, we determined which `toString` method would be executed when calling this method on the object of the analyzed class. It is always the most specific defined method

[1] https://github.com/sulir/tostring-study

– e.g., in the mentioned example, for the hierarchy `HashMap` ⇾ `AbstractMap` ⇾ `Object`, it is the method in `AbstractMap`.

We divided the results into four categories: the most specific `toString` method is defined directly in the given class, in its direct superclass, indirect superclass, or in the root of the class hierarchy (`Object`). In Fig. 1, we can see the results in a graphical form.

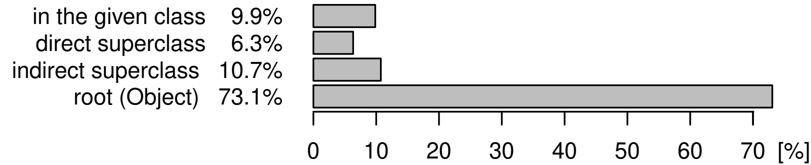

Fig. 1. The most specific toString method definition for classes

Answering **RQ1** and **RQ1.1**, only 9.9% of classes define their own `toString` implementation directly, while 17.0% derive a custom implementation from one of their superclasses. The majority of the analyzed classes (73.1%) does not define a custom string representation at all – it relies on the default one, which is useless in many cases. This highlights a need to offer fully automated generation of string representations. Although modern IDEs (integrated development environments) offer semi-automated (partially manual) generation of `toString` methods, developers evidently do not utilize this feature to a high extent.

4 ToString Invocations

In Java, `toString` methods can be called either explicitly or implicitly. An explicit call is a standard method call visible in the source code, e.g.:

```
String value = someObject.toString();
```

An implicit `toString` call is performed automatically on a non-string object during concatenation of a string and non-string expression:

```
Object nonStringObject;
String value = "string " + nonStringObject;
```

When answering **RQ2** and its sub-questions, we considered all parsable Java files (106,473). We automatically searched for all `toString()` method invocations and categorized them as either implicit or explicit. The implicit calls were recognized by searching for the “`+`” operator with one string and one non-string operand. The locations of the calls were also noted and categorized as either directly inside another `toString` method definition or outside it. Then, we selected a random subset of the calls, which we inspected manually to gain insights about string representation usage.

For the summary of quantitative results, see Table 1.

Table 1. Explicit vs. implicit toString calls, calls from other toString methods.

Call type	Explicit (`obj.toString()`)	36.7%
	Implicit (concatenation)	63.3%
Call location	From within other `toString` methods	13.0%
	Outside a `toString` method	87.0%

4.1 Explicit and Implicit Calls

First, we will answer **RQ2.1**. A majority of `toString` calls (63.3%) is implicit – i.e., not visible in the source code at a first sight. On average, we found 0.34 explicit and 0.59 implicit `toString` invocations per Java file. This totals to 0.93 `toString` invocations per Java file, which means string representations are fairly commonly utilized in the code.

4.2 Calls from Other toStrings

Answering **RQ2.2**, we determined that 13.0% of all `toString()` calls were located directly in another `toString` definition. This means string representations are fairly often built recursively from other textual representations. In addition to this, the `toString` method is sometimes called also indirectly through a chain of other auxiliary methods.

4.3 Other Usage Scenarios

Except for building string representations of objects from other string representations, there are other common usage scenarios. Now we will look at the examples of them, thus answering **RQ2.3**.

String representations are commonly used to work with exceptions. There are two recurring patterns which we encountered during the manual inspection of the results. In the first case, an object is converted to a string and then included in the message of the exception being constructed and thrown. The second case is the conversion of a caught exception to a string and passing it to a logger.

Textual representations were used for logging beyond exception handling: Any object can be converted to a string and written to a logger (e.g., a file or console) for debugging purposes.

We also encountered multiple `toString` invocations in introspection libraries, where they were used to convert runtime metadata of the program into strings, so they could be later used for debugging and visualization purposes.

A very common usage of `toString` is for a conversion of a string-like object (e.g., an `XMLString`) to a standard `String`. The most frequent conversion we encountered was from a `StringBuilder` (or related classes), which is a mutable implementation of a text string in Java.

Finally, `toString` calls were present also in unit tests. In assertions, a string representation was often compared with another string representation or a literal.

Particularly the last two applications (type conversion and unit testing) are typical examples of cases when the `toString` method is not used only for debugging and development purposes, but where it has an influence on the correct program functionality. This has an important implication for us. If we wanted to implement an automatically applied string representation generator, we would have to take great care not to break the existing representations. For example, if an object had the `toString` method implemented in an indirect superclass and we wanted to generate a more specific one directly in the given class, we could break the functionality if the program logic relied on the `toString` from the superclass.

5 ToString Contents

To answer **RQ3**, we analyzed all `toString` methods defined inside classes in our corpus (considering only the ones for which we had the source code available and superclasses resolved). In total, 11,302 method definitions were analyzed.

5.1 Language Constructs

Our main goal was to find out whether there exist certain very common forms of `toString` definitions, which are possibly repetitive. To answer **RQ3.1**, for each analyzed method, we obtained a set of all AST (abstract syntax tree) node types present in it. For example, the definition

```
return "a" + "b";
```

contains the AST node set {Return, Literal, BinaryOperator}. We excluded too generic AST types, such as blocks or type references. Then, we counted how often each such node set is present in the collection of the analyzed `toString` methods.

In Table 2, we can see a list of the most frequently occurring node sets, along with the percentages of the `toString` methods where they occur and examples of their source code. This provides us an overview of how typical `toString` methods look, which of them were probably semi-automatically generated with an IDE, and to what extent their generation could be fully automatized.

Almost 14% of `toString` methods consist of a return statement and a mix of literals, field reads on the current object (`this`), and binary operators (most notably, “`+`”). We suppose these definitions were frequently generated using an IDE.

The second most frequent set contains method invocations in addition to these node types. Here we observe less schematic code, since on some variables, various methods were called to more precisely specify the string representation.

The next three node type sets usually represent the same essence – only one member variable is included in the string representation, without any additional

Table 2. The most frequent node type sets in the collection of analyzed toString method definitions.

%	**Node type set**	**Source code example**
13.82	{Return, Literal, ThisAccess, FieldRead, BinaryOperator}	`return "PreparedStatementCreator: sql=[" +` `  sql + "]; parameters=" + this.parameters;`
8.84	{Return, Literal, ThisAccess, FieldRead, BinaryOperator, Invocation}	`return "registry[" + this.sessions.size()` `  + " sessions]";`
8.03	{Return, ThisAccess, FieldRead, Invocation}	`return table.toString();`
5.81	{Return, ThisAccess, Invocation}	`return getName();`
5.63	{Return, ThisAccess, FieldRead}	`return flag;`
5.42	{Return, Literal}	`return "Immediate key";`
5.15	{Return, ThisAccess, Invocation, BinaryOperator, Literal}	`return getArtifact() + " < "` `  + getRepositories();`
2.90	{Return, ThisAccess, FieldRead, Invocation, Literal}	`return String.format("%s[value=%s]",` `  getClass().getSimpleName(), value);`

2.62	{Return, LocalVariable, ConstructorCall, VariableRead, Invocation, Literal, ThisAccess, FieldRead}	`final StringBuilder buf`
		`  = new StringBuilder();`
		`buf.append("[local: ").append(this.local);`
		`buf.append("defaults: ")`
		`  .append(this.defaults);`
		`buf.append("]");`
		`return buf.toString();`

2.44	{Return, LocalVariable, ConstructorCall, VariableRead, Invocation, Literal, ThisAccess, FieldRead, BinaryOperator, If}	`StringBuilder sb`
		`  = new StringBuilder("FactoryCreateRule[");`
		`if (creationFactory != null) {`
		`  sb.append("creationFactory=");`
		`  sb.append(creationFactory);`
		`}`
		`sb.append("]");`
		`return (sb.toString());`

information. In cases when the given field was not a sole member variable of the class, a question arises how this field was selected and why its name is not printed.

When a `toString` method returns solely a literal, it is often related to the given class name; this is not a rule though. Methods consisting of invocations, binary operators, and literals frequently contained getters – but again, this is not a rule.

The last three lines in Table 2 are mainly just variations of the first one, but using either a string-formatting function `String.format` or mutable strings (`StringBuilder`, with or without null checking) instead of string concatenation.

5.2 Reusing Superclass Implementations

Next, we were interested in whether `toString` implementations reuse the string representation of a superclass in some way – i.e., whether they include a call to `super.toString()`. We analyzed all `toString` methods inside classes derived from a class other than `Object`.

We found out only 10.3% of such methods call `super.toString()`. This is probably caused by the fact that only a small portion of classed directly defines `toString`, which makes reuse difficult.

5.3 Schematic Implementations

Since our future vision is to fully automate the string representation generation, in **RQ3.3** we focused on the schematicity of the `toString` implementations. We define a schematic implementation as a method definition which consists only of one or more of these language constructs: `return`, `this`, string literals, `super.toString()`, direct reading of the fields of the current object, a null-checking `if` statement or ternary operator, and standard string-building operations (a `toString` call, string concatenation, a `String.format` call, basic `StringBuilder` and `StringBuffer` operations).

We found that 33.4% of `toString`s correspond to our definition of schematic implementation, while 66.6% of them are more complicated. The former group looks promising with respect to the possibility of fully automated generation.

5.4 Member Variables Read

Finally, in **RQ3.4**, we further inspect the schematic implementations found in the previous step. Schematic `toString` definitions often contain a class name and a list of name–value pairs of the object's member variables. Many of them were probably generated using a wizard in an IDE. However, note that the generation process is not fully automated: the programmer must still manually select which member variables will be included in the textual representation and which ones will be omitted. Otherwise the representations might get impractically long, especially if the member variables themselves are non-primitive objects with their

own – similarly structured – string representation. Some fields might be also considered irrelevant to the application domain (e.g., logging support). Therefore, we hypothesize only a portion of member variables are included in string representations.

To empirically confirm our intuition, we consider all schematic `toString` implementations inside classes with at least one non-constant (i.e., not static final) member variable. For them, we determine what proportion of non-constant member variables of the given class are read in its `toString` method. For simplicity, we consider only member variables (fields) defined directly in a given class, not in its superclasses.

The results are depicted in Fig. 2. About half of the analyzed classes (51.5%) read all fields in its string representation, while the other half (48.5%) include only some of them or none.

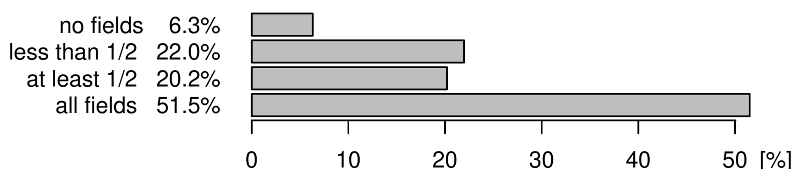

Fig. 2. The proportion of member variables read in schematic toString methods

We consider the latter case to be a suitable candidate for full automation: Using heuristics or machine learning, we could select a subset of fields which should be included in the string representation.

6 Threats to Validity

Now we will describe threats to the validity of our study according to Wohlin et al. [15].

6.1 Construct Validity

During the analysis, we excluded duplicate files based on package names and top-level type names (fully qualified class names). Nevertheless, there still may be duplicate classes present under different names.

In the first research question, we excluded test classes based on standardized directory names, which might not be sufficient. However, it is questionable whether it is useful to have string representations of test classes or not, and what exactly is considered a test class.

For the third research question, we represented methods by the sets of node types they consist of, which may be an oversimplification. The displayed examples might not fully represent the whole node type sets. Nevertheless, RQ3.1 was partially qualitative and the listed exact percentages are mainly supplementary.

Our definition of schematic implementation is rather ad-hoc. However, it was inspired by common `toString` generation templates of the most used Java IDEs.

During the analysis of member variable reads, we did not include member variables defined in the superclasses. However, this improves the clarity of the study since the fields in superclasses are not always accessible (there may be private, package-private and located in a different package, etc.), which would complicate the interpretation of the results.

6.2 External Validity

Findings about the corpus used in this study might not be generalizable, since the corpus may not be representative of all software used in practice. We tried to mitigate this threat by basing it on an existing curated corpus and including dependencies to increase its size. Although it was not constructed directly by crawling large-scale software forges such as GitHub, it includes many projects hosted on these sites. The corpus includes software maintained by many organizations and individuals, developed by thousands of developers. On the other hand, the inclusion of dependencies in the analysis might add bias as the code of libraries may be different from other code.

Although the original corpus was dated, we selected only projects updated in the last year. The list of the 15 base projects therefore includes rather mature projects – but their dependencies can include also newer and smaller ones.

7 Related Work

The most similar study to ours was performed by Schwarz [10]. He found that 28% of Smalltalk repositories in Squeaksource included a definition of a `printOn:` method (a Smalltalk analogy of `toString`), and the average length of a `printOn:` method was 7.1 lines. A few other findings regarding the properties of `printOn:` methods were very briefly described. In contrast to him, our study was performed on Java, uses a larger corpus, defines more precise research questions, and includes a more in-depth analysis.

There exist multiple works studying language and API usage in general. However, they are not specifically focused on `toString` methods in any way. For example, Dyer et al. [3] performed a study of a huge collection of AST nodes to study the usage of Java language features. Three separate studies – by Ma et al. [6], Qiu et al. [8], and Lemay [5] – studied general API call usage on large Java corpora. These three studies consistently found out that the `toString` method is one of the most called standard API methods in source code.

Xu et al. [16] parse `toString` methods to determine the mapping between lines in log files and the source code fragments that produced them. They did not perform any empirical study of the `toString` methods though.

In a study of the Hackage Haskell corpus, `deriving Show`, responsible for the string representation generation, was the most common `deriving` statement [1].

Complementary to string representations, researchers designed also approaches to specify graphical representations of objects: namely DoodleDebug [10], Vebugger [9], and the Moldable Inspector [2]. In our previous work [11], we investigated what properties developers expect from such visual representations.

8 Conclusion and Future Work

In this paper, we described an empirical study of textual object representations in Java. We found that the majority of classes (73%) relies on the default (and often not very useful) `toString` implementation defined in the `Object` class.

A majority of `toString` method calls (63%) is implicit – using a string concatenation operator with one non-string operand. The `toString` methods are called from other representation-building methods (13%), in exception handling and logging code, in introspection utilities, when performing type conversion, and from unit tests. They are therefore consumed not only by developers as a debugging aid, but are sometimes necessary for the software to function properly.

A very common `toString` definition consists of literals, field reads, and string concatenation operators. String representations of superclasses are rarely reused (10% of `toString`s contain them). A significant portion of `toString` definitions (over 33%) is rather schematic. Half of such schematic implementations read all member variables, the other half excludes some of them.

In the future, we would like to extend our study. First, we could use a larger and more representative source code corpus. Second, we would like to answer our research question in a more in-depth manner, particularly RQ3. Finally, our main future goal is fully automated generation of useful and still succinct `toString` methods without any manual interaction by a developer, particularly by selecting the most important fields to display using machine learning.

Acknowledgments. This work was supported by Project VEGA No. 1/0762/19 Interactive pattern-driven language development. This work was also supported by FEI TUKE Grant no. FEI-2018-57 "Representation of object states in a program facilitating its comprehension".

References

1. Bezirgiannis, N., Jeuring, J., Leather, S.: Usage of generic programming on hackage: Experience report. In: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming. pp. 47–52. WGP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2502488.2502494
2. Chiş, A., Nierstrasz, O., Syrel, A., Gîrba, T.: The Moldable Inspector. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 44–60. Onward! 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2814228.2814234

3. Dyer, R., Rajan, H., Nguyen, H.A., Nguyen, T.N.: Mining billions of AST nodes to study actual and potential usage of Java language features. In: Proceedings of the 36th International Conference on Software Engineering. pp. 779–790. ICSE 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568295
4. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101. MSR 2014, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2597073.2597074
5. Lemay, M.J.: Understanding Java usability by mining GitHub repositories. In: 9th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2018). OpenAccess Series in Informatics (OASIcs), vol. 67, pp. 2:1–2:9. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019). https://doi.org/10.4230/OASIcs.PLATEAU.2018.2
6. Ma, H., Amor, R., Tempero, E.: Usage patterns of the Java standard API. In: Proceedings of the XIII Asia Pacific Software Engineering Conference. pp. 342–352. APSEC '06, IEEE Computer Society, Washington, DC, USA (2006). https://doi.org/10.1109/APSEC.2006.60
7. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: A library for implementing analyses and transformations of Java source code. Softw. Pract. Exper. **46**(9), 1155–1179 (Sep 2016). https://doi.org/10.1002/spe.2346
8. Qiu, D., Li, B., Leung, H.: Understanding the API usage in Java. Inf. Softw. Technol. **73**(C), 81–100 (May 2016). https://doi.org/10.1016/j.infsof.2016.01.011
9. Rozenberg, D., Beschastnikh, I.: Templated visualization of object state with Vebugger. In: Proceedings of the 2014 Second IEEE Working Conference on Software Visualization. pp. 107–111. VISSOFT '14, IEEE Computer Society, Washington, DC, USA (2014). https://doi.org/10.1109/VISSOFT.2014.26
10. Schwarz, N.: DoodleDebug, objects should sketch themselves for code understanding. In: 5th Workshop on Dynamic Languages and Applications. DYLA 2011 (2011)
11. Sulír, M., Juhár, J.: Draw this object: A study of debugging representations. In: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming. pp. 20:1–20:11. ACM (Apr 2019). https://doi.org/10.1145/3328433.3328454
12. Sulír, M., Porubän, J.: Augmenting source code lines with sample variable values. In: Proceedings of the 2018 26th IEEE/ACM International Conference on Program Comprehension (ICPC). pp. 344–347 (May 2018). https://doi.org/10.1145/3196321.3196364
13. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: The Qualitas Corpus: A curated collection of Java code for empirical studies. In: Proceedings of the 2010 Asia Pacific Software Engineering Conference. pp. 336–345. APSEC '10, IEEE Computer Society, Washington, DC, USA (2010). https://doi.org/10.1109/APSEC.2010.46
14. Terra, R., Miranda, L.F., Valente, M.T., Bigonha, R.S.: Qualitas.class corpus: A compiled version of the Qualitas Corpus. SIGSOFT Softw. Eng. Notes **38**(5), 1–4 (Aug 2013). https://doi.org/10.1145/2507288.2507314
15. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer (2012)
16. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 117–132. SOSP '09, ACM, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629587