JFeature: Know Your Corpus

Idriss Riouak*, Görel Hedin*, Christoph Reichenbach*, and Niklas Fors*
*idriss.riouak, gorel.hedin, christoph.reichenbach, and niklas.fors (@cs.lth.se)
Department of Computer Science, Lund University, Sweden

***Abstract*—Software corpora are crucial for evaluating research artifacts and ensuring repeatability of outcomes. Corpora such as DaCapo and Defects4J provide a collection of real-world open-source projects for evaluating the robustness and performance of software tools like static analysers. *However, what do we know about these corpora? What do we know about their composition? Are they really suited for our particular problem?* We developed JFeature, an extensible static analysis tool that extracts syntactic and semantic features from Java programs, to assist developers in answering these questions. We demonstrate the potential of JFeature by applying it to four widely-used corpora in the program analysis area, and we suggest other applications, including longitudinal studies of individual Java projects and the creation of new corpora.**

***Index Terms*—Source-Code Analysis, Software Tools, Software Corpora**

I. Introduction

The impact of our research in computer science is bounded by our ability to demonstrate and communicate how effective our techniques and theories really are. For research on software tools, the dominant methodology for demonstrating effectiveness is to apply these tool to "real-life" software development tasks and to measure how well they perform. Blackburn et al. [3] outline this process in considerable detail, highlighting the need for *appropriate experimental design* (to construct experiments), *relevant workloads* (to obtain relevant data from the experiments), and *rigorous analysis* (to obtain rigorously justified insights from experimental data). The strength of our insights is then bounded by the weakest link in this chain.

Carefully curated, pre-packaged workloads such as the DaCapo Benchmark suite [2], Defects4J [20], the Qualitas Corpus [30], and XCorpus [7] can help ensure that we use relevant workloads. However, no software corpus aims to be representative of all software, and for any given research question there may not be any one corpus designed to answer that question, so we must still validate that the corpus we choose is relevant to what we want to show.

For instance, the DaCapo corpus aims to provide benchmarks with "more complex code, richer object behaviors, and more demanding memory system requirements" [2] than the corpora that preceded it, and it systematically demonstrates complex interactions between architecture and the Java Run-Time Environment, whereas Defects4J collects "real bugs to enable reproducible studies in software testing research" [20]. Despite DaCapo's focus on run-time performance and Defects4J's focus on software testing, both suites have seen heavy use in research that they were not explicitly intended for, including the authors' own work in static analysis [8], [28] (using Defects4J), and in compilers [9] and dynamic invariant checking [27] (for DaCapo).

For each of these ostensible mis-uses, the authors selected the corresponding benchmark corpus as the highest-quality corpus they were aware of whose original purpose seemed sufficiently close to the intended experiments. This divergence between research question and corpus purpose required the authors to carefully re-validate the subset of the corpus that they had selected by hand.

In this paper, we argue that there is a need for increased automation and decision support for selecting benchmarks for specific research questions, and present JFeature, a static analysis tool designed to help researchers in this process. JFeature identifies how often a Java project uses key Java features that are significant for different types of software tools. JFeature operates at the source code level, and is capable of identifying not only local syntactic features that may be challenging to encode in regular expression search tools like `grep`, but also complex semantic features that depend on types and libraries. We have implemented JFeature in the JastAdd [11] ecosystem as an extension of the ExtendJ [9] Java Compiler. This implementation architecture gives easy access to types and other properties computed by the compiler, and also supports extensibility, allowing researchers to adapt the analysis to fit their specific needs.

We demonstrate JFeature by running it on several widely-used corpora, specifically the DaCapo, Defects4J, Qualitas, and XCorpus corpora.

Our main contributions are:

- JFeature as an example of a tool for extracting information about the features used in Java source code, and
- An overview over JFeature's key insights on the DaCapo, Defects4J, Qualitas, and XCorpus corpora.

The rest of this paper is organised as follows: Section II introduces JFeature and discusses the design decisions that underpin the tool. Section III shows the results of applying JFeature to the four corpora. Section IV illustrates how JFeature can be extended to extract new features, taking advantage of properties in the underlying Java compiler. Section V outlines future applications of JFeature. Section VI discusses related work, and Section VII summarizes our conclusions.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

II. JFeature: automatic feature extraction

We have designed JFeature as an extension of the ExtendJ extensible Java compiler. ExtendJ is implemented using Reference Attribute Grammars (RAGs) [10] in the JastAdd metacompilation system. ExtendJ is a full Java compiler, feature-compliant for Java 4 to 7 and close to being feature-compliant for Java 8[1]. In building compilers by means of attribute grammars [21], the abstract syntax tree (AST) is annotated with properties called *attributes* whose values are defined using equations over other attributes in the AST. RAGs extend traditional attribute grammars by supporting that attributes can be links to other AST nodes. ExtendJ annotates the AST with attributes that are used for checking compile-time errors and for generating bytecode. Example attributes include links from variable uses to declarations, links from classes to superclasses, types of expressions, etc. These attributes are exploited by JFeature to easily identify AST nodes that match a particular feature of interest.

A. Java version features

There are many different features that could be interesting to investigate in a corpus. As the default for JFeature, we have defined feature sets for different versions of Java, according to the Java Language Specification (JLS). A user can then run JFeature to, e.g., investigate if a corpus is sufficiently new, or select only certain projects in a corpus, based on what features they use. If desired, a user can extend the feature set for a specific purpose.

In recent years there have been several new releases of the Java language. Currently, Java 18 is the latest version available. However, most projects utilise Java 8 or Java 11, both of which are long-term support releases (LTS).

Table I summarises the main features introduced in each Java release after the initial release (JDK 1.0) up to Java 8. We have classified the features into either

- `Syntactic`: can be identified using a context-free grammar, or
- `Semantic`: additionally needs context-dependent information such as nesting structure, name lookup, or types.

While most features are syntactic, there are several features that are semantic, and where the attributes available in the compiler are very useful for identification of the features.

Given any Java 8 project, JFeature collects all the feature usages, grouped by release version. By default, JFeature supports twenty-six features[2], but users may extend the tool and add their own. We have chosen these features by looking at each Java release note [12]–[19]. We included features that represent the most significant release enhancements, i.e., libraries or native language constructs whose use significantly impacts program semantics. In particular, we included the usage of two libraries, JAVA.UTIL.CONCURRENT.* and JAVA.LANG.REFLECT.*, because their usage may be pertinent for the evaluation of academic static analysis tools.

[1]https://extendj.org/compliance.html

[2]The complete implementation can be found at https://github.com/lu-cs-sde/JFeature.

Feature		Kind	
		Syn	Sem
Java 1.1 - 4, 1997-2002 – [12]–[15]			
Inner Class			✓
java.lang.reflect.*			✓
Strictfp		✓	
Assert Stmt		✓	
Java 5, 2004 – [16], [17]			
Annotated Compilation Unit		✓	
Annotations	Use	✓	
	Decl	✓	
Enum	Use		✓
	Decl	✓	
Generics	Method	✓	
	Constructor	✓	
	Class	✓	
	Interface	✓	
Enhanced For		✓	
Varargs		✓	
Static Import		✓	
java.util.concurrent.*			✓
Java 7, 2011– [18]			
Diamond Operator		✓	
String in Switch			✓
Try with Resources		✓	
Multi Catch		✓	
Java 8, 2014– [19]			
Lambda Expression		✓	
Constructor Reference			✓
Method Reference			✓
Intersection Cast		✓	
Default Method		✓	

TABLE I: Major changes in the Java language up to Java 8.

B. Collecting features

To collect features, JFeature uses *collection attributes* [4], [25], also supported by JastAdd. Collection attributes aggregate information by combining contributions that can come from anywhere in the AST. A *contribution clause* is associated with an AST node type, and defines information to be included, possibly conditionally, in a particular collection. Both the information and the condition can be defined by using attributes.

For JFeature, we use a collection attribute, `features`, on the root of the AST. The value of `features` is a set of objects, each defined by a contribution clause somewhere in the AST. The objects are of type `Feature` that models essential information about the extracted feature: the Java version, feature name, and absolute path of the compilation unit where the feature was found.

Figure 1 shows an example with JastAdd code at the top of the figure, and below that, an example program and its attributed AST. The `features` collection is defined on the nonterminal `Program`, which is the root of the AST (line 1). Then two features are defined, STRICTFP and STRING IN SWITCH (lines 3-5 and 7-9).

STRICTFP is a syntactic feature that corresponds to the modifier `strictfp`. In ExtendJ, modifiers are represented by the nonterminal `Modifiers` which contains a list of modifier keywords, e.g., `public`, `static`, `strictfp`, etc. To find

```
1  coll HashSet<Feature> Program.features();
2
3  Modifiers contributes
4    new Feature("JAVA2", "Strictfp", getCU().path())
5    when isStrictfp() to Program.features();
6
7  Switch contributes
8    new Feature("JAVA7", "StringInSwitch", getCU().path())
9    when getExpr().type().isString() to Program.features();
```

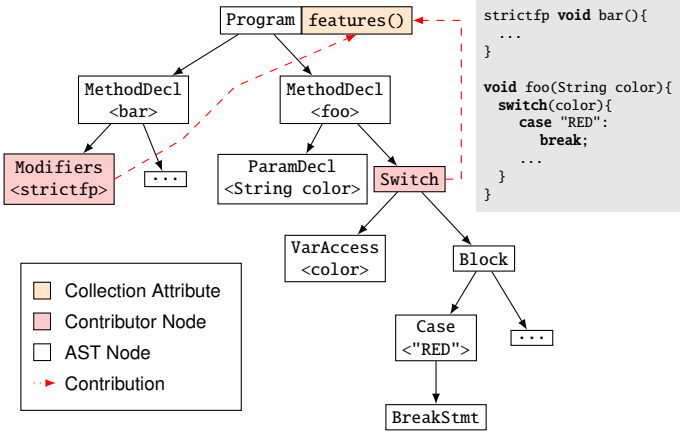

Fig. 1: Example definitions of features.

out if one of the keywords is `strictfp`, ExtendJ defines a boolean attribute `isStrictfp` for `Modifiers`. To identify the STRICTFP feature, a contribution clause is defined on the nonterminal `Modifiers` (line 3), and the `isStrictfp` attribute is used for conditionally adding the feature to the collection (line 5). The absolute path is computed using other attributes in ExtendJ: `getCU` is a reference to the AST node for the enclosing compilation unit, and `path` is the absolute path name for that compilation unit (line 4).

STRING IN SWITCH is a semantic feature in that it depends on the type of the switch expression. It cannot be identified with simple local AST queries or regular expressions. Here, the contribution clause is defined on the nonterminal `Switch`, and the feature is conditionally added if the type of the switch expression is a string. ExtendJ attributes used here are `type` which is a reference to the expression's type, and `isString` which is a boolean attribute on types.

III. CORPORA ANALYSIS

We used JFeature to analyse four widely used corpora, to investigate to what extent the different Java features from Table I are used. We picked the newest available version of each of the corpora.

A. Corpora Description

DaCapo Benchmark Suite: Blackburn et al. introduced it in 2006 as a set of general-purpose (i.e., library), freely available, real-world Java applications. They provided performance measurements and workload characteristics, such as object size distributions, allocation rates and live sizes. Even if the primary goal of the DaCapo Benchmark Suite is intended as a corpus for Java benchmarking, there are several instances of frontend and static analysers evaluation. For evaluation, we used version 9.12-bach-MR1 released in 2018.

Defects4J: introduced by Just et al., is a bug database consisting of 835 real-world bugs from 17 widely-used open-source Java projects. Each bug is provided with a test suite and at least one failing test case that triggers the bug. Defects4J found many uses in the program analysis and repair community. For evaluation, we used version 2.0.0 released in 2020.

Qualitas Corpus: is a set of 112 open-source Java programs, characterised by different sizes and belonging to different application domains. The corpus was specially designed for empirical software engineering research and static analysis. For evaluation, we used the release from 2013 (20130901).

XCorpus: is a corpus of modern real Java programs with an explicit goal of being a target for analysing dynamic proxies. XCorpus provides a set of 76 executable, real-world Java programs, including a subset of 70 programs from the Qualitas Corpus. The corpus was designed to overcome a lack of a sufficiently large and general corpus to validate static and dynamic analysis artefacts. The six additional projects added in the XCorpus make use of dynamic language features, i.e., invocation handler. For evaluation, we used the release from 2017.

B. Evaluation

Methodology: To compute complete semantic analysis with JFeature and ExtendJ, all dependent libraries and the classpath are needed for each analysed project. Unfortunately, different projects use different conventions and build systems, making automatic extraction of this information difficult. Therefore, for our study of the full corpora, we decided to extract features depending only on the language constructs and the standard library, but that did not require analysis of the project dependencies. This way, we could run JFeature on these projects without any classpath (except for the default standard library).

Table II shows an overview of the results of the analysis. For each corpus, we report the number of projects that use a particular feature from Table I. More detailed results, including the results for all 26 features, and counts for each individual project, are available at https://github.com/lu-cs-sde/JFeature/blob/main/features.xlsx.

For standard libraries, like `java.lang.reflect.*` and `java.util.concurrent.*`, we count all variable accesses, variable declarations, and method calls whose type is hosted in the respective package.

While ExtendJ mostly complies to the JLS version 8, its Java 8 type inference support diverges from the specification in several corner cases. As Table II shows, these limitations did not affect DaCapo, but they did surface in 43 method calls in 9 projects (2 projects in Defects4J that we manually inspected to validate our findings.

TABLE II: Corpora Analysis. Each entry represents the total number of projects utilising the respective feature.

Corpus (# Projects)	JAVA 1.1 - 4				JAVA 5													JAVA 7				JAVA 8				
	Inner Class	java.lang.reflect.*	Strictfp	Assert Stmt	Annotated CU	Annotation		Enum		Generics				Enhanced For	VarArgs	Static Import	java.util.concurrent.*	Diamond Operator	String in Switch	Try w/ Resources	Multi Catch	Lambda Expression	Constructor Reference	Method Reference	Intersection Cast	Default Method
						Use	Decl	Use	Decl	Method	Constructor	Class	Interface													
DaCapo (15)	15	12	2	5	0	8	4	14	8	6	2	7	4	8	7	5	7	2	1	3	2	2	0	2	0	0
Defects4J (16)	16	15	1	8	0	15	7	16	14	13	3	12	10	15	13	14	14	14	7	13	10	10	5	8	1	1
Qualitas (112)	109	100	4	51	9	67	35	109	45	55	7	59	41	68	49	46	50	1	1	1	1	0	0	0	0	0
XCorpus (76)	74	65	4	28	3	39	21	74	32	31	4	35	22	39	28	25	27	4	2	3	3	2	1	2	0	0

Corpus	Projects															
	Mock		Asm		Derby		JUnit		Tomcat		Xerces		JRep		JMeter	
	1.1	2.0	3.3	5.2	10.14	10.9	4.10	4.12	6.0	7.0	2.8	2.10	1.1	3.7	2.5	3.1
DaCapo			✓		✓			✓	✓		✓					
Defects4J	✓															
Qualitas						✓	✓			✓		✓		✓	✓	
XCorpus	✓		✓							✓		✓	✓			✓

TABLE III: Projects used in the corpora with different versions.

Corpora overlap: Figure 2 shows the overlap between the four corpora as two Venn diagrams where each number represents a project. In the left diagram, two versions of the same project are counted as two separate projects. In the right diagram, we only consider the project name, disregarding the version. From the left diagram, we can see that Defects4J does not overlap with any other corpus analysed. As expected, most of the projects are shared across Qualitas and XCorpus as XCorpus was built as an extension of Qualitas. From the diagrams, we can see that eight projects (145-137) are used among the corpora, but with different versions. Table III details these projects and versions.

Discussion: Table II provides insight into the features utilised by each project. Using Qualitas Corpus as an illustration, we see that `strictfp` is only used in four projects. Similarly, in DaCapo, fewer than fifty percent of the projects use concurrency libraries. With JFeature, we can achieve a fine-grained classification of the properties. We can, for instance, distinguish between uses and declarations of annotations, and when it comes to generics, we can distinguish between the declarations of generic methods, classes, and interfaces, providing the user with a better comprehension of the corpus. It is apparent that most projects utilise only Java 4 and Java 5 features. With the exception of Defects4J, few projects employ Java 7 and Java 8. Indeed, this table reveals that Defects4J is the most modern corpus, as nine of the fourteen assessed applications utilise at least one of the observed Java 8 features.

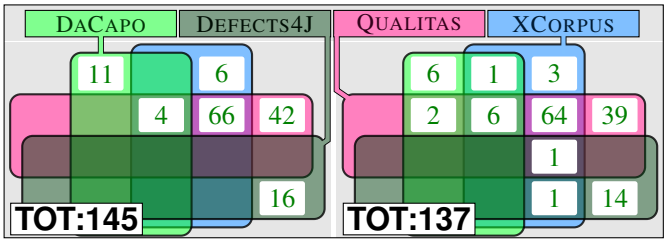

Fig. 2: Project overlap. In the left diagram, two projects with the same name but different versions are counted as distinct—the diagram to the right shows overlap when versions are disregarded.

IV. Extensibility

Extensibility is one of the key characteristics of JFeature. Users can create new queries to extract additional features, making use of all attributes available in the ExtendJ compiler. We illustrate this by adding a new feature, Overloading, that measures the number of overloaded methods in the source code. Listing 1 shows the JastAdd code for this: we define a new boolean attribute, `isOverloading`, that checks if a method is overloaded. We then use this attribute to conditionally contribute to the `features` collection, only for overloaded method declarations. The attribute `isOverloading` is defined using several ExtendJ attributes: the attribute `hostType` is a reference to the enclosing type declaration of the method declaration. A type declaration, in turn, has an attribute `methodsNameMap` that holds references to all methods for that type declaration, both local and inherited. If there is more than one method for a certain name, that name is overloaded.

Listing 1: Definition of the Overloading feature

```
MethodDecl contributes
  new Feature("JAVA1", "Overloading", getCU().path())
  when isOverloading() to Program.features();

syn boolean MethodDecl.isOverloading()
  = hostType().methodsNameMap().get(getID()).size() > 1;
```

For the computation to work, it is necessary to supply the classpath, so that ExtendJ can find the classfiles for any direct or indirect supertypes of types in the analysed source code. We analysed 16 distinct projects for which we successfully extracted the classpaths and dependencies required for ExtendJ compilation. The results provided by JFeature for the sixteen

Projects	∼ KLOC	Number of Methods	Overloaded Methods	%
antlr-2.7.2	34	2081	358	17,2
commons-cli-1.5.0	6	585	76	13
commons-codec-1.16-rc1	24	1812	422	23,3
commons-compress-1.21	71	5359	571	10,7
commons-csv-1.90	8	716	93	13
commons-jxpath-1.13	24	2030	167	8,23
commons-math-3.6.1	100	7229	1779	24,6
fop-0.95	102	8317	666	8,01
gson-2.90	25	2289	125	5,46
jackson-core-2.13.2	48	3687	839	22,8
jackson-dataformat-2.13	15	1122	161	14,3
jfreechart-1.0.0	95	6980	1000	14,3
joda-time-2.10	86	9324	1257	13,5
jsoup-1.14	25	2556	408	16
mockito-4.5.1	19	2054	318	15,5
pmd-4.2.5	60	5324	1021	19,2

TABLE IV: Results from the Overloading feature.

projects are summarised in Table IV. As can be seen, each project has overloaded methods. In some cases, such as `commons-codec`, `commons-math`, and `jackson-core`, more than one fifth of the methods are overloaded.

Overloading is a good example of a feature that requires semantic analysis—it can not be computed by a simple pattern match using regular expressions or a context-free grammar.

V. Use cases for JFeature

We already discussed two possible use cases for JFeature: corpus evaluation (Section III), and extending JFeature to identify specific features (Section IV). In this section, we discuss two additional use cases: longitudinal studies and project mining.

A. Longitudinal Study

JFeature can be used to conduct longitudinal studies, i.e., changes occurring over time. As an example, we conducted a study on Mockito and its evolution on the adaption of Java 8 features over time. Mockito is one of the most popular Java mocking frameworks and has an extensive history with over 5,000 commits. Java 6 was utilised by Mockito until version 2.9.x. With version 3.0.0, Java 8 was adopted. The evolution of the occurrences of Lambda Expressions and Try With Resources is depicted in Figure 3. As can be seen, at commit

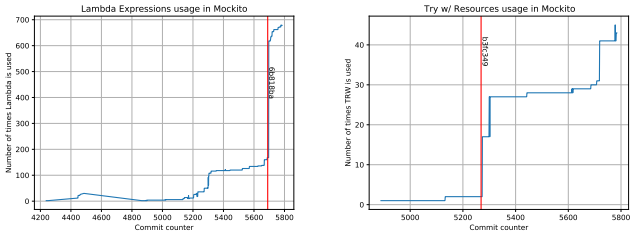

Fig. 3: Usage of Lambda Expressions and Try With Resources in Mockito over time.

number 5269[3], there is a substantial increase in utilisation of try with resources, whereas at commit number 5696[4], there is a significant increase in the use of lambda expressions.

B. Project mining

Contemporary revision control hosting services (GitHub[5], GitLab[6], bitbucket[7]) offer uniform interfaces to the source code of millions of software projects. These interfaces enable researchers to "mine" software projects at scale, filtering by certain predefined properties (e.g., the number of users following the project or the main programming language). For example, the *GitHub Java Corpus* [1] collects almost 15,000 projects from GitHub, filtered to only include Java projects that have been forked at least once. Combining JFeature with these query mechanisms allows researchers to select projects by more detailed syntactic and semantic features. For instance, a corpus suitable for answering questions about race detection [22] could select projects that make explicit use of java.util.concurrent.*, while an exploration of functional programming patterns [6] could select projects that use Lambda Expressions and Method References.

VI. Related work

Existing tools for code metrics are usually focused on code quality metrics, rather than what language features are used, and typically analyse the intermediate representation rather than the source code. One example is the CKJM tool [29] for the Chidamber and Kemerer metrics [5]. Another example, that more closely resembles ours, is jCT, an extensible metrics extractor for Java 6 IL-Bytecode, introduced by Lumpe et al. [23], in 2011. Like us, they evaluated their tool on Qualitas Corpus; however, because jCT works only on annotated bytecode and not on source code, the number of features that can be extracted is limited. A significant amount of information is lost during the compilation of Java source code to Java bytecode. For example, enhanced `for` statements, diamond operators and certain annotations, such as `@Override`, are not present in the bytecode. For XCorpus, the authors analysed the language features used, and a summary was presented in their paper [7]. They also analysed the bytecode, which was implemented using the visitor pattern.

A way to improve the user experience would be to integrate JFeature with a visualisation tool like *Explora* [26]. The idea behind *Explora* is to provide to the user a visualisation tool designed for simultaneous analysis of multiple metrics in software corpora. Finally, JFeature may be enhanced by incorporating automated dependency extractors, such as MagpieBridge's *JavaProjectService* [24], to infer and download libraries automatically. Currently, JavaProjectService infers the dependencies for projects using *Gradle* or *Maven* as build system.

[3]Commit: b3fc349.
[4]Commit: 6b818ba.
[5]https://github.com
[6]https://gitlab.com
[7]https://bitbucket.org

VII. Conclusions

We have presented JFeature, a declarative and extensible static analysis tool for the Java programming language that extracts syntactic and semantic features. JFeature comes with twenty-six predefined queries and can be easily extended with new ones.

We ran JFeature on four widely used corpora: the DaCapo Benchmark Suite, Defects4J, Qualitas Corpus, and XCorpus. We have seen that, among the corpora, Java 1-5 features are predominant. This leads us to conclude that some of the corpora may be less suited for the evaluation of tools that address features in Java 7 and 8.

We have illustrated how JFeature can be extended to capture semantically complex features by writing the queries as attribute grammars, extending a full Java compiler. This allows powerful queries to be written that can make use of all the compile-time properties computed by the compiler.

We discussed several possible use cases for JFeature: evaluation of corpora, mining software collections to create new corpora, and longitudinal studies of how projects have evolved with regard to the use of language features. We also note that for some features to be analysed, the full classpath and dependencies are required. An interesting future direction is therefore to combine JFeature with recent tools that support automatic extraction of such information from projects that follow common build conventions.

References

[1] M. Allamanis and C. Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[3] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[4] J. T. Boyland. *Descriptional composition of compiler components*. PhD thesis, University of California, Berkeley, 1996.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[6] D. R. Cok. Reasoning about functional programming in java and c++. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 37–39, 2018.

[7] J. Dietrich, H. Schole, L. Sui, and E. D. Tempero. XCorpus - An executable Corpus of Java Programs. *Journal of Object Technology*, 16(4):1:1–24, 2017.

[8] A. Dura, C. Reichenbach, and E. Söderberg. JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. In *Proceedings of the ACM on Programming Languages*. ACM, Sep 2021.

[9] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 1–18, 2007.

[10] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[11] G. Hedin and E. Magnusson. JastAdd - an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[12] JDK 1.1 New Feature Summary. Note: Available as file jdk1.1.8/docs/relnotes/features.html in zip file for Java Development Kit (JDK) Documentation 1.1 (jdk118-doc.zip) at https://www.oracle.com/java/technologies/java-archive-downloads-javase11-downloads.html (login needed), Last accessed: 2022-08-03.

[13] Java 2 SDK, Standard Edition, version 1.2. Summary of New Features and Enhancements. Note: Available as file jdk1.2.2.202/docs/relnotes/features.html in zip file for Java 2 SDK, Standard Edition Documentation 1.2.2_006 (jdk-1_2_2_006-doc.zip) at https://www.oracle.com/java/technologies/java-archive-javase-v12-downloads.html (login needed), Last accessed: 2022-08-03.

[14] Java 2 SDK, Standard Edition, version 1.3. Summary of New Features and Enhancements. Note: Available as file docs/relnotes/features.html in zip file for Java 2 SDK, Standard Edition Documentation 1.3.1 (java1.3.zip) at https://www.oracle.com/java/technologies/java-archive-13docs-downloads.html (login needed), Last accessed: 2022-08-03.

[15] *Java 2 Sdk for Solaris Developer's Guide*. Sun Microsystems, 2000. ISBN: 978-14-005-2241-5, Note: Includes description of New Features and Enhancements for Java 1.4.

[16] New Features and Enhancements. J2SE 5.0. https://docs.oracle.com/javase/1.5.0/docs/relnotes/features.html. Accessed: 2022-08-03.

[17] Java SE 6 Features and Enhancements. https://www.oracle.com/java/technologies/javase/features.html. Accessed: 2022-08-03.

[18] Java SE 7 Features and Enhancements. https://www.oracle.com/java/technologies/javase/jdk7-relnotes.html. Accessed: 2022-08-03.

[19] What's New in JDK 8. https://www.oracle.com/java/technologies/javase/8-whats-new.html. Accessed: 2022-08-03.

[20] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[21] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[22] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. Residual investigation: Predictive and precise bug detection. *ACM Trans. Softw. Eng. Methodol.*, 24(2):7:1–7:32, Dec 2014.

[23] M. Lumpe, S. Mahmud, and O. Goloshchapova. jCT: A Java Code Tomograph. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 616–619, 2011.

[24] L. Luo, J. Dolby, and E. Bodden. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In A. F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[25] E. Magnusson, T. Ekman, and G. Hedin. Extending attribute grammars with collection attributes–evaluation and applications. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 69–80, 2007.

[26] L. Merino, M. Lungu, and O. Nierstrasz. Explora: A visualisation tool for metric analysis of software corpora. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 195–199. IEEE, 2015.

[27] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently?: A language for heap assertions at GC time. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 256–269. ACM, 2010.

[28] I. Riouak, C. Reichenbach, G. Hedin, and N. Fors. A precise framework for source-level control-flow analysis. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–11. IEEE, 2021.

[29] D. Spinellis. Tool writing: A forgotten art? *IEEE Software*, 22(4):9–11, July/August 2005.

[30] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, Dec. 2010.