Understanding the Syntactic Rule Usage in Java

Dong Qiu[a], Bixin Li[a,*], Earl T. Barr[b], Zhendong Su[c]

[a] *School of Computer Science and Engineering, Southeast University, China*
[b] *Department of Computer Science, University College London, UK*
[c] *Department of Computer Science, University of California Davis, USA*

Abstract

Context: Syntax is fundamental to any programming language: syntax defines valid programs. In the 1970s, computer scientists rigorously and empirically studied programming languages to guide and inform language design. Since then, language design has been artistic, driven by the aesthetic concerns and intuitions of language architects. Despite recent studies on small sets of selected language features, we lack a comprehensive, quantitative, empirical analysis of how modern, real-world source code exercises the syntax of its programming language.
Objective: This study aims to understand how programming language syntax is employed in actual development and explore their potential applications based on the results of syntax usage analysis.
Method: We present our results on the first such study on Java, a modern, mature, and widely-used programming language. Our corpus contains over 5,000 open-source Java projects, totalling 150 million source lines of code (SLoC). We study both independent (*i.e.* applications of a single syntax rule) and dependent (*i.e.* applications of multiple syntax rules) rule usage, and quantify their impact over time and project size.
Results: Our study provides detailed quantitative information and yields insight, particularly (i) confirming the conventional wisdom that the usage of syntax rules is Zipfian; (ii) showing that the adoption of new rules and their impact on the usage of pre-existing rules vary significantly over time; and (iii) showing that rule usage is highly contextual.
Conclusions: Our findings suggest potential applications across language design, code suggestion and completion, automatic syntactic sugaring, and language restriction.

Keywords: Language syntax, empirical study, practical language usage

1. Introduction

Syntax and semantics define a programming language. Informally, a language has many features. A language's syntactic rules provide the most direct means to measure the use of a language's features. Thousands of programming languages exist; each embodies a different set of possible language features. Language designers usually have limited knowledge on how programmers actually use a language [1]. This leads to many unnatural and rarely used features being introduced, while expected ones not introduced [2, 3]. In addition, many language features, especially language syntax, remain a significant barrier to novice programmers [4, 5].

We tackle the question of how to systematically understand these features and their usage. Rather than ad-hoc characterizations of features, we propose the use of language grammars to precisely and systematically characterize language features. Indeed, most programming language features quite directly map onto syntactic constructs. Therefore, we study how programmers use language features by analyzing their use of the language syntax.

Knuth conducted the first study to understand how programmers use `FORTRAN` over 40 years ago [1]. Similar studies were subsequently performed on `COBOL` [6, 7], `APL` [8] and `Pascal` [9] between the 1970s and 1980s. In recent decades, there has been little quantitative study demonstrating how a modern programming language is used in practice, especially from the perspective of language syntax. Previous studies have investigated the use of subsets of language features (*e.g.*, Java generics [10] and Java reflection [11]). Although Dyer *et al.* [12] investigated the use of newly-introduced features over three main language releases, they only examined a relatively small subset of language features and did not consider pre-existing features.

Studying how a large number of real-world programs use language syntax may help validate or disprove the many popular "theories" about what language features are most popular, most useful, easiest to use, *etc.* that abound in popular literature about programming and on the Internet. In addition, the gap between language features and their actual usage may guide pedagogy, giving teachers insight into how to teach a programming language in a better way. Language designers may leverage data on actual syntactic rule usage to optimize the design of languages, *e.g.* simplifying unpopular features or identifying boilerplate that could be eliminated. We will provide concrete examples when presenting our detailed study results.

[*] Corresponding author.
Email addresses: `dongqiu@seu.edu.cn` (Dong Qiu), `bx.li@seu.edu.cn` (Bixin Li), `e.barr@ucl.ac.uk` (Earl T. Barr), `su@cs.ucdavis.edu` (Zhendong Su)

To this end, we perform a large-scale empirical study on a diverse corpus of over $5,000$ real-world Java projects to gain insight into how syntactic rules are used in practice. We generate abstract syntax trees (ASTs) for approximately 150 million SLoC, and tabulate and analyze the occurrences of all syntactic rules. In particular, to understand how syntax rules are used over time, we have checked out over $13,000$ versions from the studied projects' revision histories to understand rule usage evolution.

We also perform depth-2 bounded nesting analysis to investigate dependent rule usage. Indeed, when using a grammar to parse a string, some nonterminals in the grammar can be reached only after another nonterminal has been traversed. For $X, Y \in N$, the set of nonterminals, and $\alpha, \beta \in (N \cup T)^*$ where T is the set of terminals, we write $X \overset{*}{\to} \alpha Y \beta$ to denote that Y *depends* on X. We bound this dependency because, in the limit, all nonterminals vacuously depend on the grammar's start symbol. In this work, we consider $k = 2$ and report our dependency results for $X \overset{2}{\to} \alpha Y \beta$, as these short range dependencies are closer to the sentences that programmers write and think about and thus are better candidates for identifying idioms.

In summary, this paper makes the following contributions:

- It presents the first effort in 30 years to conduct a large-scale, comprehensive, empirical analysis of the use of language constructs in a modern programming language, namely Java;
- This work is the first to study dependent rule usage and quantify its contextual nature. This is also the first to study the evolution of rule usage over time, the adoption of new rules, and how new rules impact the usage of pre-existing ones.
- The results show that: (i) 20% of the most-used rules account for 85% of all rule usage, while 65% of the least-used rules are used $< 5\%$ of the time and 40% only $< 1\%$ of the time; (ii) 16.7% of the rules are unpopular and are adopted in $< 25\%$ of the projects (*e.g.* `assert` statement, `labeled` statement, and `empty` statement); and (iii) for dependent rule usage, 6% of the combinations exhibit strong dependency with $> 50\%$ probability.

Taken together, our results permit language designers to empirically consider whether new constructs are likely to be worth the cost of their implementation and deployment. They also identify boilerplate (*i.e.* repetitive rule usage) that new constructs may profitably replace. For example, we have observed a reduced use of anonymous class declarations, while an increased use of the enhanced-for constructs *w.r.t.* all syntactic rule usage. We believe that work like ours enables data-driven language design, analogous to how Cocke's study at IBM in the 1970s on the actual usage of CISC instructions eventually led to the RISC architectures.

2. Study Design and Results

This section describes our methodology in detail, with special attention given to the study subject and the research questions, followed by our general findings.

2.1. Study Subject

Java Syntax. To understand how programmers adopt syntax, we selected Java, a modern, mature and widely-used programming language as our research subject. Java's syntax is the set of rules defining how a Java program is written and interpreted; it is essentially a dialect of C/C++. Major releases of the Java Language Specification (JLS) track its constant evolution.

In this paper, we survey 132 syntactic rules in total, distributed in JLS1∼JLS4[1] [13–16]. Table 1 lists the distribution, including the release date and corresponding updates. In contrast to the study by Dyer *et al.* [12], which focuses on the newly imported language syntax rules, we concentrate on the *complete* set of the syntactic rules. The details of the rules can be found online[2].

Table 1: Overview and evolution of the JLSs.

Version	Release Date	#Added Rules	#Updated Rules
JLS1	1996	115	-
JLS2	2000	4	-
JLS3	2005	12	16
JLS4	2013	1	2

Code Corpus. Our corpus is a large (around 150 million SLoC) collection of open-source real-world Java programs containing $5,646$ projects retrieved from Github, one of the most popular repositories. The projects were selected based on their popularity (*i.e.* size of *watchers*, *stars* and *forks*). The corpus contains not only widely-used Java projects maintained by reputable open-source organizations (*e.g.* Tomcat, Hadoop, Derby from the Apache Software Foundation and JDT, PDT, EGIT from the Eclipse Foundation), but also small projects developed by novice programmers. All these projects are managed by Git, one of the most popular version control systems in the open-source community. Table 2 provides summary statistics on the corpus.

The corpus is also diverse, covering projects of different size and development history. It contains small, medium and large projects, where the number of Java files within projects ranges from 1 to $39,247$. The corpus also includes projects with short, medium and long lifecycles, where their development years span from 1 to 17 and the commits with each repository range from 1 to $123,938$. The corpus thus provides a wide and comprehensive range of projects on which to study the evolution of syntactic rule usage.

Many projects contain duplicate source files. Such duplications probably distort the statistics of the syntactic rule usage. To tackle this problem, we take the following measures. For each Git repository, we only analyze the *main* branch to avoid the redundant computation on a project with multiple copies. Also the duplicated files may still exist within a project. A search program we wrote helps us to automatically identify such functionally equivalent source files. Our tool only detects type-1 file clone (which may differ in whitespace, comments and layout).

[1] For simplicity, JLS1, JLS2, JLS3 and JLS4 are used to represent the 1st edition, 2nd edition, 3rd edition and Java SE 7 edition of the JLS, respectively.

[2] It is available at: http://dong-qiu.github.io/papers/lang_syntax/appendix.pdf

Table 2: Summary statistics on the Java code corpus.

Corpus Summary	
Repository	Github
# of Projects	$5,646$
# of Files	$1,392,528$
Lines of Code	$144,081,228$
Project Scale Range (# of files)	1∼39, 247
Project History Range (# of years)	1∼17
Project Commits Range (# of commits)	1∼123, 938

Tool Support. We developed a tool, named the *J*ava *S*yntactic *R*ule Extractor (JSRExtractor), that collects all the syntactic rules from our source code corpus. JSRExtractor uses Eclipse EGit [17] to interact with Git project repositories and automatically check out source code versions. It leverages the Eclipse JDT [18] parser, which parses Java code and builds its abstract syntax tree (AST). The tool integrates Neo4j [19], a popular graph database, to store and manage the extracted ASTs. It can quickly traverse ASTs and obtain the usage of syntactic rules, including their dependency usage. For instance, JSRExtractor supports performing a depth-2 bounded search on ASTs to calculate syntactic rule dependencies. To analyze the distribution of syntactic rule usage over time, we used JSRExtractor to check out multiple versions of each project's source code to study the year-by-year evolution of syntax rule usage. Based on this highly optimized and well tested tool, processing the whole corpus took approximately 2 days of computing on a dual-Xeon server with 16GB of main memory.

2.2. Research Questions and Key Findings

This study aims to answer how language syntax is adopted in real open-source projects. To this end, we designed three specific research questions (RQ) for investigation. This section also lists summary results to provide an overview.

RQ1: *How are syntactic rules used in practice?*

For each project, we determine how many syntactic rules it used. In addition, we compute the *popularity* of each rule, *i.e.*, how much of the projects use it. We are also interested in the concrete usage of different syntax rules within the code corpus. This data tells us which syntactic rules are most used and which are least used, highlighting some *unpopular* rules as possible candidates for language designers to continuously improve their designs, and finally achieve the goal of simplifying the language to ease its maintenance, especially to reduce its cognitive load on developers.

Findings: (i) The usage of syntax rules obeys Zipf's law [20]: some rules are used frequently, while others rarely; (ii) most projects use only a subset of syntax rules; and (iii) project size, measured in size of Java files, correlates with the adoption of syntactic rules. Section 3 explains these findings in detail.

RQ2: *How are syntactic rules used in practice over time?*

This RQ sheds light on the *evolution* of the syntactic rule usage. For each project, we investigate the frequency of syntactic rules changed during the project's development life-cycle. For each rule, we report its historical adoption rate by projects. The data shows syntactic rules becoming *popular* or *unpopular*. We are also interested in understanding how the usage of evolved syntactic rules after language updates, *e.g.*, how the new, or updated rules impact the usage of pre-existing rules?

Findings: (i) The use of most existing syntactic rules remains stable over time, with some exceptions whose usage is declining; (ii) most newly introduced rules were adopted by developers gradually but some have been widely used in projects. However, not all of them were used as expected; and (iii) newly added rules do impact the use of the existing relevant rules. Section 4 details our findings.

RQ3: *How strongly do rule usage in practice depend on context?*

In contrast to RQ1, which studies the syntactic rules in isolation, this RQ helps us understand syntactic rule usage dependencies in real-world code, and answer questions like "What rules tend to follow a certain type of "parent" rule?" In particular, we calculate the conditional probability of dependent rule usage to investigate what rules are likely to be adopted together.

Findings: (i) Syntactic rules exhibit nontrivial dependency (*e.g.*, 6% of rule combinations show strong dependency with $> 50\%$ probability); and (ii) rule usage is contextual and helps identify potential syntactic sugar to simplify a language or guide syntactic (rather than lexical) refactoring or code completion and suggestion. Section 5 explains our findings in detail.

3. Single Rule Usage in Practice

Notation. First, we formalize the measures that we use. Each project $P_i = \{f_1, f_2, \cdots\}$ is a set of files. Our source code corpus is a set of projects: $C = \{P_1, P_2, P_3, \cdots\}$. When r_i is a syntactic rule, $R = \{r_1, r_2, \cdots, r_n\}$ is the set of syntactic rules under analysis. We use $O(P_i)$ to denote the multiset of rules used in project P_i. We let m_X denote multiplicity function of the multiset X; the multiplicity $m_{O(P)}(r)$ returns the multiplicity, *i.e.* the count of uses, of the rule $r \in P$. We elide X, when its binding is clear from context. $R(P)$ denotes the set that underlies $O(P)$ whose indicator function returns 1 for every rule in $O(P)$ with multiplicity > 0. Likewise, we use multiset $O(f)$ to record the usage of rules in the file and $R(f)$ to denote $O(f)$'s underlying set, for $f \in P$.

3.1. Aggregate Results

From the perspective of syntactic rule, we study how rules are used in our corpus, tallying their *popularity* and *frequency*. To this end, we defined three measures:

1. *PP(r)* represents the *P*ercentage of the *P*rojects that adopt the syntactic rule r:

$$PP(r) = \frac{|C'|}{|C|}, \text{where } C' = \{P_i \mid r \in R(P_i)\}$$

PP(r) measures a rule's *popularity* across the projects in a corpus. The value is larger when more projects adopt r; when all projects in a corpus adopt r, *PP(r) = 1.*

2. *PF(r)* represents the *P*ercentage of the *F*iles that adopt the syntactic rule r:

$$PF(r) = \frac{\sum\limits_{P_i \in C} |P_i'|}{\sum\limits_{P_i \in C} |P_i|} \; (P_i' \subseteq P_i, \forall f_i \in P_i', r \in R(f_i))$$

   PF(r) measures the *popularity* of rule use across files, which is a finer-grained perspective to measure the rule adoption. The value is larger when more files adopt r; when all files adopt r, *PF(r) = 1.*

3. *PO(r)* is the *P*ercentage of *O*ccurrences of the rule r computed as the count of uses of r over the count of all rule uses:

$$PO(r) = \frac{\sum\limits_{P \in C} m_{O(P)}(r)}{\sum\limits_{r \in R} \sum\limits_{P \in C} m_{O(P)}(r)}$$

   PO(r) evaluates the rule use *frequency* among the corpus. The value would be larger if the occurrences of rule r is higher in the source code.

From the perspective of the software projects, we further wonder how projects use the syntactic rules. Namely, how much of the syntactic rules are enough to construct a project in common? In addition, from a finer-grained angle, we wish to learn how much of the syntactic rules are used to construct a file in common. To this end, we define two values to measure:

4. *PSRP(P$_i$)* represents the *P*ercentage of the *S*yntactic *R*ules that are used in Project P_i. *PSRP(P$_i$)* evaluates *efficiency* of the current language grammars. The value would be larger if more syntactic rules are adopted in project P_i. When all syntactic rules are adopted in P_i, *PSRP(P$_i$) = 1.*

$$PSRP(P_i) = \frac{|R(P_i)|}{|R|}$$

5. *PSRF(f$_i$)* represents the *P*ercentage of the *S*yntactic *R*ules that are used in File f_i. *PSRF(f$_i$)* also evaluates the grammar *efficiency* from a finer-grained perspective. The value would be larger if more syntactic rules are adopted in file f_i. When all syntactic rules are adopted in f_i, *PSRF(f$_i$) = 1.*

$$PSRF(f_i) = \frac{|R(f_i)|}{|R|}$$

To investigate how a single syntax rule is used in practice, we calculated *PP*, *PF*, *PSRP* and *PSRF* values based on the *latest* snapshot of every repository. Figure 1 shows the distributions of these values. Regarding the *PP* values in Figure 1(a), most of the rules were adopted in around 50%∼80% of the projects. More than half of the rules (53.8%) were used in over 75% of the projects, in which only 3 rules (`compilation-unit`, `class declaration` and `identifier`) were used in all projects. Instead, the *PP* value of some syntactic rules were quite small. More than 15% of the syntactic rules were adopted in less than 25% projects (*e.g.* the `label` statement). Regarding the *PF* values, most of the rules were adopted 5%∼30% of the files. A tiny subset of the rules (9%) were used in over 75% of the files. The same 3 rules were also indispensable to construct a Java file.

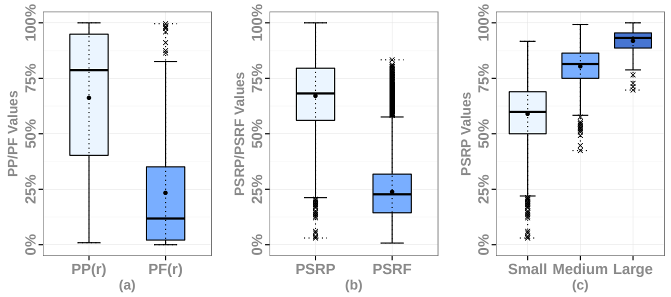

Figure 1: Boxplot of the syntactic rule usage. (a) shows the distribution of PP/PF values ; (b) shows the distribution of PSRP/PSRF values; (c) shows the impact on PSRP values by project scale.

Regarding the *PSRP* value in Figure 1(b), *not all* the syntactic rules are adopted in the project development. Programmers usually employed a subset of the rules to develop the software projects. Most projects adopted around 60%∼80% of all syntactic rules. Small group of the projects (13.8%) adopted over 85% of the rules, in which only 3 projects used the complete set of the rules. In contrast, 16.5% of the projects adopted only less than 50% of the syntactic rules. Some outlier projects in our corpus used extremely few rules. For instance, the project *Templatebread* developed by *haxzamatic* only adopted 8 syntactic rules. Regarding the *PSRF* value, around 20%∼30% of the syntactic rules were adopted to construct a Java file. Only 3% of the files required more than 50% of the rules. The file with maximum *PSRF* value covered 85% of the syntactic rules.

It is natural to speculate the *PSRP* values are tightly related with the project scales. Our hypothesis is that projects with larger scale adopt more syntactic rules since large-scale systems usually involve more diverse characteristics, which requires more language features, *i.e.* syntactic rules to complete its functionalities. To verify our intuitive conjecture, we classified the corpus into three groups[3] and analyzed the distribution separately. Figure 1(c) confirms the results. Small-scale projects adopt around 60% of all syntactic rules. Projects in the medium-scale groups use around 80% of all rules. Large-scale projects do not adopt all rules, in which around 10% of the rules are not adopted. The results also confirm that syntactic rules are selectively used.

It is also interesting to validate whether the syntactic rule usage obey the Pareto principle. Figure 2(a) shows that a small number of syntactic rules account for most rule usage. The top 20% of the rules account for 85% all rule usage. The heavy tail

[3] We use the number of Java files to measure the project size. The projects with less than 100 Java files belong to the *small-scale* group. The projects containing between 100 and 1000 Java files belong to the *medium-scale* group. The rest projects belong to the *large-scale* group.

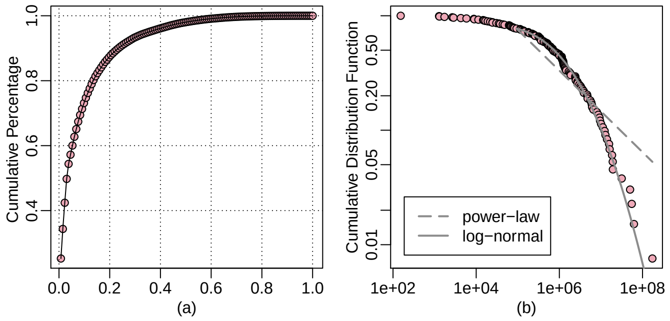

Figure 2: Rule usage distribution based on the PO values. (a) shows the cumulative percentage of the rule usage; (b) shows the cumulative distribution function of the rule usage. The points are data, solid line is best-fit log-normal and the dashed line is best-fit power-law.

covers many unpopular syntactic rules, 65% of the least-used rules account for less 5% of all rules usage; the 40% for less 1%. The heavy tail in our data indicates the possibility for optimizing the design of the language syntax, *e.g.* removing or refactoring the extremely unpopular syntax, instead of adding new syntax continuously. We also fit the rule usage data to some candidate distributions. Figure 2(b) shows that the log-normal distribution has a much better fit than the power-law distribution [21].

3.2. In-Depth Study of Interesting Rules

To further investigate the usage of the syntactic rules, we present the *PP*, *PF* and *PO* values of most representative rules in detail, which are divided into four groups: *declaration*, *statement*, *expression* and *type/annotation*. Table 3 shows the results.

Declaration. Regarding the group of the *declaration* related rules, the `class`, `method`, `import` and `package` declarations appeared in (almost) every project. They also had high probability of appearing in Java files. It is obvious as the `class` and `method` declaration are fundamental rules to implement the essential functionalities. The `import` declaration is the basic rule to introduce external functionalities, together with the `method` invocation. The `package` declaration is also indispensable as it assigns namespaces for programs to prevent name conflicts. The `constructor` declaration also had very high *PP* value, but it only appeared in about half of the Java files. Contrarily, the `annotation-type` and `enum` declaration were used much less than other declarations. This might be because both of the rules were introduced in JLS3, which caused developers have less time to be familiar with and adopt them. We also found that the *PF* and *PO* values of `annotation-type`, `anonymous-class`, `enum` and `interface` declarations had big decreases *w.r.t. PP* values. It indicates that although these rules were adopted in many projects, the use frequencies were quite low.

Statement. Regarding the group of the *statement* related rules, the `expression` statement was the most commonly used rule, which had both high *PP* and *PF* values. The usage is as expected since it is in charge of wrapping expressions into statements. The `variable-declaration`, `return` and `if` statements were posteriori frequently used where the first rule is the essential way to generate variables (in which most are objects, the key elements in Object Oriented Programming) and the last two rules are both fundamental constructs to affect the control flow of the program's execution. The `do`, `empty`, `labeled` and `type-declaration` statements were used much fewer than other statements.

Expression. Regarding the group of the *expression* related rules, the *PP* value of most rules reached over 80%, except the `super-field` access. The `method` invocation was heavily used as expected since it is the key mechanism to introduce internal and external functionalities. More than half of the files adopted the `class-instance` creation, `assignment` and `infix` expression. Other rules were used in 10%$\sim$30% of the files.

Type. Regarding the group of the *type* related rules, the usage of the `simple` type and `primitive` type covered a large portion. The `parameterized` type, which was introduced in *JLS3* to represent the generic type, was also used considerably. Conversely, the `union` type and the `wildcard` type were used much less currently, although the `wildcard` type had been adopted in many projects. In the *annotation* group, `marker` annotation was used much more than the `normal` annotation and `single-member` annotation. The data from [12] also confirmed that the top two used annotations, *i.e.* `@Override` and `@Test`, which covered more than 55% usage of all annotations, were all `marker` annotations.

In addition, we have some other interesting findings from the following three perspectives:

Rule Use Preferences

1. The `do`, `while`, `for` and `enhanced-for` statement are alternative loop statements in Java. The usage of `for` statement is 2.5 times the usage of `while` statement, and 23 times the usage of `do` statement. The `enhanced-for` statement is a *syntactic sugar* designed for `for` statement, which was introduced in *JLS3*. Its *PP* and *PF* values exceeded the corresponding values of the `for` statement and its *PO* value also approached `for`'s *PO* value. This may be because the `enhanced-for` statement is recommended for programmers to adopt over the `for` statement in practice [22]. We discuss more on the evolution of these rules in the *RQ2*.

2. The `if` and `switch` statement are both decision-making syntactic rules. The usage of `if` statement is 40 times the usage of `switch` statement. This might be caused by the restriction of applying `switch` statement. In addition, a `switch` statement involves 6 `switch` cases in average from the *PO* values, which indicates that `switch` statement was applied in the complex scenarios with more selective execution paths. The `conditional` expression is the short form of the `if` statement as the conditional operator behaves like a simple if-else. From the *PO* values, the usage of the `conditional` expression was 1/15 of the `if` statement. Using the `conditional` expression to implement the structure with multiple decisions would reduce the readability of source code.

3. The anonymous and local class are two special kinds of inner classes in the nested classes. The `anonymous-class` decla-

Table 3: Usage of the main syntactic rules.

Syntactic Rules	PP(r)	PF(r)	PO(r)
Declarations			
Annotation Type Declaration	28.9%	0.9%	0.0045%
Anonymous Class Declaration	80.7%	11.1%	0.1448%
Class Declaration	100%	87.5%	0.5147%
Constructor Declaration	96.1%	50.8%	0.3909%
Enum Declaration	54.5%	2.7%	0.0147%
Field Declaration	99.3%	63.0%	1.8740%
Import Declaration	99.9%	86.4%	3.3996%
Interface Declaration	72.7%	11.6%	0.0587%
Method Declaration	99.9%	91.1%	3.9261%
Package Declaration	99.0%	97.5%	0.4353%
Statements			
Assert Statement	24.4%	1.7%	0.0286%
Break Statement	74.1%	7.2%	0.2212%
Continue Statement	52.9%	3.6%	0.0430%
Do Statement	35.3%	1.1%	0.0115%
Empty Statement	26.6%	0.6%	0.0051%
Enhanced For Statement	83.5%	15.2%	0.2113%
For Statement	82.7%	15.0%	0.2583%
If Statement	98.6%	47.4%	2.6462%
Labeled Statement	16.4%	0.5%	0.0076%
Return Statement	98.9%	67.2%	2.8310%
Switch Statement	64.9%	5.0%	0.0677%
Synchronized Statement	39.2%	2.1%	0.0420%
Throw Statement	81.4%	19.3%	0.3777%
Try Statement	90.8%	21.5%	0.3645%
Type Declaration Statement	9.2%	0.2%	0.0018%
Variable Declaration Statement	99.5%	58.3%	3.7576%
While Statement	77.6%	9.7%	0.1077%
Expressions			
Array Access	82.9%	14.5%	0.7587%
Array Creation	82.9%	15.9%	0.3019%
Assignment	99.1%	58.3%	3.0832%
Cast Expression	94.8%	30.3%	0.8407%
Class Instance Creation	99.5%	60.4%	2.4410%
Conditional Expression	78.9%	12.5%	0.1718%
Constructor Invocation	59.3%	4.9%	0.0356%
Field Access	91.8%	32.5%	1.1060%
Infix Expression	99.1%	54.4%	4.9584%
Instanceof Expression	69.7%	11.8%	0.1872%
Method Invocation	99.9%	75.0%	16.8744%
Parenthesized Expression	92.0%	24.4%	0.8323%
Postfix Expression	83.8%	15.7%	0.3504%
Prefix Expression	93.4%	28.0%	0.7396%
Super Constructor Invocation	83.6%	22.9%	0.1558%
Super Field Access	6.8%	0.2%	0.0027%
Super Method Invocation	78.6%	11.8%	0.1282%
This Expression	96.3%	38.8%	1.4251%
Variable Declaration Expression	82.4%	14.6%	0.2394%
Types & Annotations			
Array Type	92.2%	29.0%	0.9293%
Parameterized Type	93.4%	40.2%	1.4523%
Simple Type	99.9%	95.9%	15.3843%
Union Type	2.7%	0.1%	0.0004%
Wildcard Type	64.3%	8.4%	0.1598%
Marker Annotation	98.0%	48.5%	1.1805%
Normal Annotation	56.2%	5.7%	0.1100%
Single Member Annotation	82.2%	11.5%	0.1298%

ration and `type-declaration` statement correspond to them separately in the syntactic rules. From the *PO* values, the `anonymous-class` declaration was used far beyond (83 times) the `type-declaration` statement. In most cases, the adoption of the `anonymous-class` declaration concentrated on implementing the `Listerner` or `Runable` interfaces.

4. The `infix`, `prefix` and `postfix` expressions are three different formats to compute values where the last two rules can only adopt unary operators. The `infix` expression covered most cases. The usage of `prefix` expression is 2 times than the usage of `postfix` expression as more unary operators can be used in the `prefix` expression.

Software Shapes

1. The `type` declaration[4] involved around 3 `field` declarations, 7 `method` declarations and 6 `import` declarations in average, which indicated the approximate structure and scale of a Java class; The `enum` declaration involved 6 `enum-constant` declaration averagely in practice;

2. 41.9% of classes did not have any declared constructor based on the *PF* value of the `class` declaration and `constructor` declaration. They used default constructors that were automatically generated without defining any constructors.

3. The `constructor` declaration invokes existing constructors or super constructors to improve the code reuse in general. According to the *PO* values of `constructor` declaration, `constructor` invocation, and `super-constructor` invocation, we found that 9% of the `constructor` declaration reused constructors and 40% reused super constructors.

4. 13.6% of the Java files did not involve any `import` declaration. One possible situation is that all the types used were fully qualified by programmers; Another is that these files were independent, without relying on any external functionalities.

Bad Practice

1. Based on the *PF* values, 97.5% of the Java files contained the `package` declaration. That means the remaining 2.5% of the Java files exhibited the bad practice of failing to specify their `package` declarations[5].

[4] The Java type declaration can be normal class declaration, enum declaration and interface declaration in JLS4.

[5] Java packages group related types, providing access protection and name space management. In general, types are suggested to belong in named packages.

Findings:

1. The usage of syntax rules is Zipfian; It fits the log-normal distribution.
2. Not *all* the syntactic rules were adopted to construct a project simultaneously. Project scale correlates with the adoption of the syntactic rules.
3. Statistics hidden among the syntactic rule usage data are interesting, including programmer's preferences on rule usage (*e.g.* `if` *v.s.* `switch` statement), software shapes (*e.g.* the constitution of a Java class) and bad practices (*e.g.* no package declaration is specified in a Java file)

4. Rule Usage Over Time

In *RQ1*, we analyzed the use of the syntactic rules on a single version of each project, *i.e.* the most recent one in our corpus. Here, we study their use over time. To better understand the use evolution of the syntactic rules under the project lifecycle, we would answer the following questions: (i) how did the usage of the existing syntactic rules evolve over time; (ii) how did the usage of the newly added syntactic rules evolve after they were introduced; (iii) how did the newly-introduced rules impact the usage of the existing rules. To this end, we checked out multiple snapshots of the source code for each project in the corpus and calculated the same measurements in *RQ1* on multiple versions. To facilitate observing the change trend of the evolution data, we selected *one year* as the time interval to sample the multiple snapshots.

4.1. Evolution of Existing Rules

JLS1 was the baseline version of the Java language. It published 115 syntactic rules, which still covered 87% of the rules in Java syntax even if the JLS has moved forward to *JLS4*. It is interesting to learn if the use of syntactic rules defined in *JLS1* continue the domination after new rules had been introduced. To this end, we calculated the *PO* value of syntactic rule set for different JLS versions. Table 4 shows the results based on the following equation. It is obvious the use of syntactic rules mainly converge at *JLS1*, although the value of *PO(R_{JLS1})* shrunk by 4.5% in 2014. The use of rules in *JLS3* steadily increased over time. The rules proposed in *JLS2* and *JLS4* were still rarely used.

$$PO(R_S \subseteq R) = \frac{\sum\limits_{r \in R_S} \sum\limits_{P \in C} m_{O(P)}(r)}{\sum\limits_{r \in R} \sum\limits_{P \in C} m_{O(P)}(r)}$$

To measure the change trend of the rule use, we adopt CAGR (*C*ompound *A*nnual *G*rowth *R*ate) to calculate the smoothed change rate per year within the given time period [23]. Taking *PP* value as an example, suppose *PP(r, t)* is the *PP(r)* value at time point t, we use $PP(r, t_1, t_2)$ to represent the change rate of *PP(r)* per year between time points t_1 and t_2. Hence, we have

$$CAGR_{PP}(r, t_1, t_2) = (\frac{PP(r, t_2)}{PP(r, t_1)})^{(\frac{1}{t_2 - t_1})} - 1$$

Table 4: The use of rules grouped by JLS versions.

Year	2002	2006	2010	2014
$PO(R_{JLS1})$	99.98%	99.67%	97.45%	95.42%
$PO(R_{JLS2})$	0.02%	0.01%	0.02%	0.02%
$PO(R_{JLS3})$	0%	0.32%	2.52%	4.55%
$PO(R_{JLS4})$	0%	0%	0%	0.01%

We applied the $CAGR_{PP}$ to evaluate the change of rule *popularity*, and the $CAGR_{PO}$ to evaluate the change of rule use *frequency*. We chose 2004 as the start year, and 2014 as the end year to calculate both of the values. Figure 3 presents the results. Considering the $CAGR_{PP}$ value, 27% (31/115) of the syntactic rules sustained growth over the 10 years, though their growth rates per year were slow. This might be because most of the rules are *popular* among the developers and their *PP* values essentially remained over 80%, which reduced their growth space. The $CAGR_{PP}$ value of 73% of the rules were negative, in which 42% of the rules dropped by less than 2%. Around 15% of the rules were down over 5% per year. Figure 4(a) shows some cases with rapid decline on PP values. The `empty` statement dropped steadily, fell from 71% to 27%, a total decrease of 162% in 10 years. The `synchronized` statement, the `labeled` statement and the `do` statement also had the similar downward trend. The `synchronized` statement is specific for multi-threaded applications, which restrict its usage. The `labeled` statement is always a controversial rule although it would not suffer from the problems caused by the `goto` statement [24][25]. Developers still had many debates [26], which greatly affect using this rule. For the `do` statement, we discuss more below.

Regarding the $CAGR_{PO}$, 26% (30/115) of the syntactic rules sustained growth over the 10 years, in which some of the rules increased by more than 5%, *e.g.* the `anonymous-class` declaration. When combining the $CAGR_{PP}$ and $CAGR_{PO}$, we found that 24% (28/115) of the rules have contrary change trend in which half of them (14/115) had positive $CAGR_{PP}$ values and negative $CAGR_{PO}$ values. This means although they were adopted in more proportion of projects, the occurrence proportion in the source code dropped. The `package` declaration is the case. The remaining 12 rules had negative $CAGR_{PP}$ values and positive $CAGR_{PO}$ values. The `switch` statement is the case.

4.2. Adoption of New Rules

JLS3 was a major release that introduced many important features, including *generic*, *enumeration* and *annotation*. Compared to *JLS3*, *JLS2* and *JLS4* were small update releases. As *JLS4* was released not long ago, we concentrate on the syntactic rules introduced in *JLS2* and *JLS3*.

Annotations were used to provide metadata for source code. They have no direct effect on the execution of the code they annotate. All annotation related rules were introduced in *JLS3*, involving the `annotation-type` declaration and three types of annotations, *i.e.* the `normal` annotation, the `marker` annotation and the `single-member` annotation. To measure the acceptance among the projects, we monitored the *PP* values of these rules for 10 years. Figure 4(b) shows their evolution after 2004. The *PP*

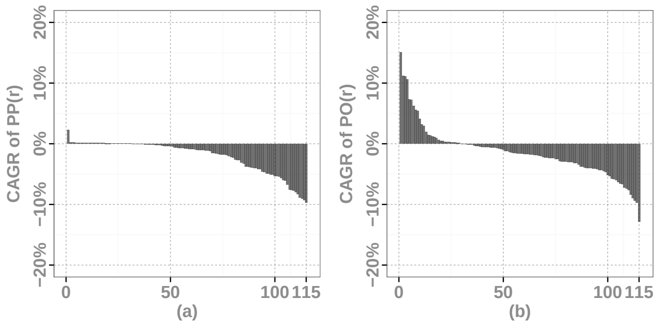

Figure 3: The change of the syntactic rule usage. (a) shows the distribution of the CAGR$_{\text{PP}}$ values; (b) shows the distribution of the CAGR$_{\text{PO}}$ values. The *x-axes* in both subfigures list every syntactic rule r defined in JLS1, and the *y-axes* represent corresponding CAGR value of each rule.

values of three annotations grew fast from 2006 and exceeded 50% in 2014. The `marker` annotation was always the most popular annotation in use, which was adopted in almost all (98%) projects in 2014. The use growth of the `single-member` annotation and the `normal` annotation were slower, but their *PP* values still achieved 82% and 56%, separately. In contrast, the use growth of `annotation-type` declaration was much slower than the annotations. This means developers preferred to use existing annotations instead of defining a new annotation type.

Certainly, *not all* the newly-added syntactic rules were used as expected. The `assert` statement is the case. The assertion enables the developers to verify their code. After the `assert` statement was introduced in *JLS2*, its *PP* values stayed between 20% and 40%. Many projects did not adopt it at all. The possible reason is that using the `assert` statement has many restrictions [27].

4.3. Impact of New Rules on Existing Rules

The `enhanced-for` statement is a popular syntactic rule introduced in JLS3. As an alternative option for `for` statement and other loop-related rules, *e.g.* `while` statement and `do` statement, it simplifies the code by its simple structure, which can be considered as a *syntactic sugar*. To look into how the `enhanced-for` statement impact the use of other loop-related rules, we calculate the PO_L for each loop rule based on the following equation where $L = \{r_{for}, r_{while}, r_{do}, r_{enhanced\text{-}for}\}$.

$$PO_L(r \in L) = \frac{\sum\limits_{P \in C} m_{O(P)}(r)}{\sum\limits_{r \in L} \sum\limits_{P \in C} m_{O(P)}(r)}$$

Figure 4(c) shows the evolution of the four rules. It depicts that the PO_L value of `enhanced-for` statement increased from 0% to 36%, which caused the decrease of the PO_L value of the `for` statement (from 70% to 43%) and the `while` statement (from 31% to 18%). The usage of `enhanced-for` statement already approached the usage `for` statement. The possible reason is that `enhanced-for` statement allows you to iterate through a collection without having to create an Iterator or without having to calculate beginning and end conditions for a counter variable, which reduces writing the repeated code for developers and makes the code easier to read and understand. In addition, the PO_L values of `do` statement stayed fairly low all the time. It *never* became popular among the loop-related rules. As the decrease of the `while` statement in use, the `for` and the `enhanced-for` statement covered almost 80% usage. We calculate the *correlation coefficient* (cc) between the `enhanced-for` and other loop rules. The table below shows the results, which indicates that `enhanced-for` and `for`, `enhanced-for` and `while` are inversely correlated.

`enhanced-for`	`for`	$cc = -0.989$
`enhanced-for`	`while`	$cc = -0.948$
`enhanced-for`	`do`	$cc = 0.021$

Generic mechanism is a significant update in JLS3. It enables types (classes and interfaces) to be parameters when defining classes, interfaces and methods. The generic code has many benefits, including provide strong type checks at compile time and eliminate the use of casts [15]. From the PP values of the `parameterized` type (99.0% in 2014), the *generic* has been adopted in most projects. To further understand the use of the generic in classes, interfaces and methods, we look into the *generic* adoption in related generic-enabled rules, *e.g.* the `class` declaration and the `class-instance` creation. To this end, we calculate the proportion of the rules that apply the generic. Figure 4(d) shows the results. The use of generic `class-instance` creation reached 12.5%, which was higher than the use of three kinds of *generic* declarations. The use of the generic type (including `parameterized` type and `wildcard` type in Java syntax) also approached 10%, *w.r.t.* all `simple` types. The use proportion of the generic `interface` declaration is 3 times than the use of the generic `class` declaration. Instead, the use of the generic `method` declaration and the generic `method` invocation remained relatively low, compared to the other rules. On the whole, the use *generic* maintained rapid growth. In addition, as the code using *generic* can reduce the usage of casts, we confirmed this benefit from the *PO* value of `cast` expression, which decreased from 1.69% (in 2004) to 0.63% (in 2014).

Findings:

1. The use of most existing syntactic rules remained stable. Some of the rules were losing their attraction, *e.g.* the `empty`, `labeled` and `do` statement;
2. Most newly-introduced rules were adopted by programmers gradually and some have been widely used in projects (*e.g.* the `marker` annotation). Exceptions also existed (*e.g.* the `assert` statement).
3. The newly-added rules did impact the use of the existing relevant rules. The `enhanced-for` statement greatly reduced the use of other *loop* related rules.

5. Dependent Rule Usage

The use of syntactic rules is dependent. In practice, we might have the experience that the `switch` statement is often applied

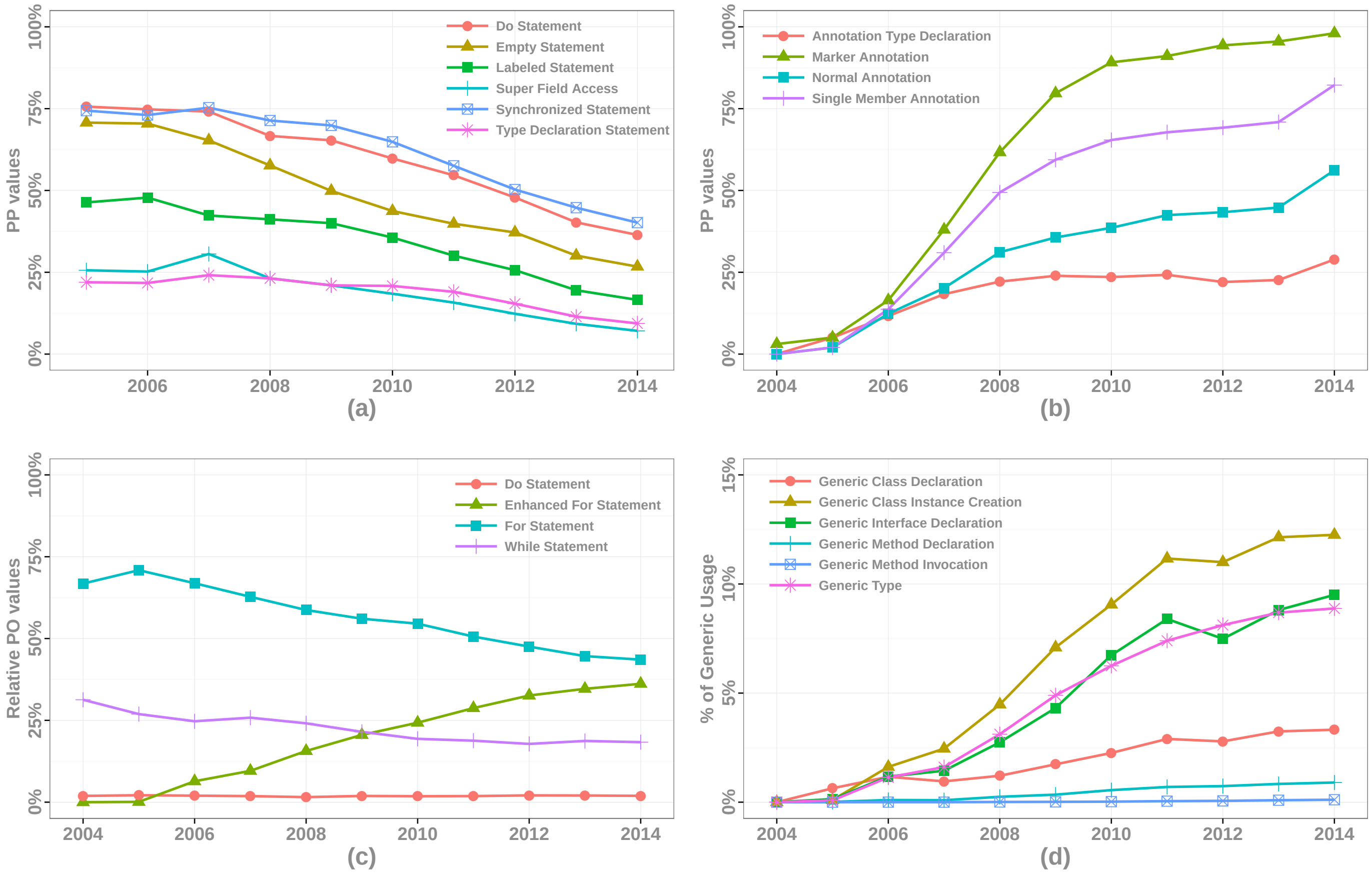

Figure 4: The evolution of the rule usage across ten years. (a) shows the change trend of the existing *unpopular* rules where its *y-axis* represents the PP value; (b) shows the change trend of the annotation-related rules where its *y-axis* represents the PP value; (c) shows the change trend of the loop-related rules where its *y-axis* represents the $\mathrm{PO_L}$ values; (d) shows change trend of the generic-related rules where its *y-axis* represents the percentage of the *generic* usage.

under the `while` statement, but rarely under the `for` statement. Another accessible example is that nested loops are often observed from the source code. However, no study has focused on this aspect. Hence, it is significant to investigate *dependencies* among syntactic rules.

In *RQ1* and *RQ2*, we have analyzed the use of syntactic rules individually. In *RQ3*, we analyze syntax rule dependencies and various nesting depth constraints in real-world code. To this end, we use bounded depth rule dependencies to capture contextual dependencies. As discussed earlier in Section 1, we consider depth 2. That is, for a pair of rules (r_1, r_2), if r_1 is fired, we calculate how likely r_2 is fired. Hence, we compute the conditional probability $Pr(r_2|r_1)$ to express depth-2 dependencies. Figure 5 gives a concrete example for calculating $Pr(r_2|r_1)$.

5.1. Aggregate Results

We calculate $Pr(r_2|r_1)$ for all possible usage dependencies among 132 syntactic rules. Figure 6 shows the heat map of the rule usage dependencies, which illustrates that strong dependencies do exist.

Overall, we have found 1273 kinds of usage dependencies with $Pr(r_2|r_1) > 0$, which cover 7% of all possible combinations. Around 6% (80/1273) of the usage dependencies have $Pr(r_2|r_1) > 50\%$. In addition, we discover that 23% (31/132) of the syntactic rules are used dependently. Figure 7 provides the depth-2 dependencies of some interesting syntactic rules. For each selected rule r_1, we only list five rules r_2 that have higher $Pr(r_2|r_1)$.

Besides nested loops, which we consider in detail later, we have also found some interesting nesting cases: (i) over $1/5$ of the `if` statements contain `if` statements in their bodies; (ii) In 13.54% of `method` invocations, other `method` invocations are nested in either the caller expressions or the method arguments; (iii) 1.8% of the `class` declaration contain another `class` declaration.

Returning `null` is considered a bad practice in most cases except when `null` is the expected result under certain conditions [22, 28]. We have found that 5.4% of the `return` statements return null. We next present some interesting examples based on $Pr(r_2|r_1)$ values.

5.2. Case Studies

Exception Handling. In Java, errors are usually managed by an exception object. The syntactic rule `try` statement is used to handle exceptions. We have found that 13.1% of the `try` statements adopt the *try-finally* structure, without using a

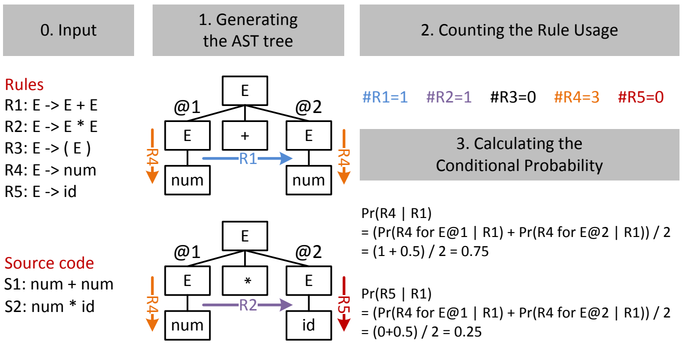

Figure 5: An example of calculating the rule use dependency.

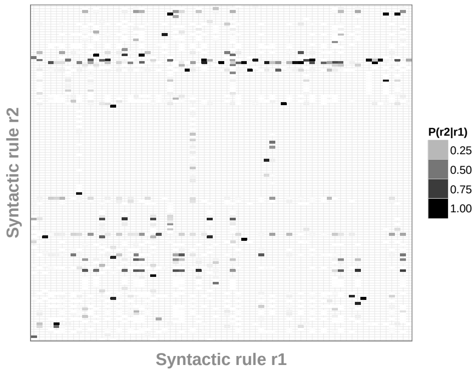

Figure 6: The heat map of the rule use dependency.

catch-clause. Such usage is reasonable when the programmer cannot handle the exception locally and must clean up the resources when the exception is triggered.

Within the use of the standard *try-catch-finally* structure, 27% of the try statements catch exceptions and re-throw them from a catch-clause, where 85% of the exceptions are wrapped before being thrown. We also find that 16% of catch-clause does nothing in handling exceptions. The behavior of catching an exception and ignoring it is usually a bad practice [22, *Item 65*].

Nested Loop. A loop is a fundamental structure that allows code to be repeatedly executed. A nested loop is a loop inside the body of another loop, which is often applied to more complex structures, *e.g.* accessing a matrix. It is interesting to study the probability of using nested loops in practice. Hence, we calculate $P(r_2|r_1)$ where $r_1, r_2 \in \{r_{for}, r_{while}, r_{do}, r_{enhanced-for}\}$. The following table shows the results. Around 10% of the for statements are nested loops, containing other loop-related rules, most of which are for statements. For enhanced-for statements, while statements and do statements, 7%∼8% of the rules contain other looping structures. Like for statements, the enhanced-for statements and while statements usually contain themselves as inner loops. In contrast, do statements favor one of the other three rules. In total, 8.77% of the loops contain nested loops.

$Pr(r_2 \mid r_1)$		r_2				
		for	enhanced-for	while	do	**total**
r_1	for	8.94%	0.73%	0.88%	0.10%	10.65%
	enhanced-for	1.09%	5.71%	0.46%	0.02%	7.28%
	while	2.52%	1.40%	2.93%	0.43%	7.28%
	do	2.71%	2.12%	2.16%	0.67%	7.66%

Enhanced-for Loop. Before the enhanced-for statement was introduced, programmers usually select the for statement to iterate over a range of values. When we traverse a list called *items*, we would likey use the following code:

```
for(int i=0; i<items.size(); i++){...}
```

It is natural that when developers adopt the for statement, the expression in the for initialization part is usually fired by the variable-declaration statement (int i=0); the for termination part is usually fired by infix expression (i<items.size()) and the for update part is usually fired by postfix (or prefix) expression (i++). The data in Figure 7 confirm our intuition. The use of the variable-declaration, infix, prefix and postfix expressions in the for statement covers 93.1%, and their proportion is close to 1:1:1. The data indicate a rule usage pattern that the variable-declaration, infix, postfix and postfix expressions are bundled with the for statement. This provides strong evidence for language designers to construct a *new* syntactic rule (sugar) to facilitate coding. Hence, the enhanced-for statement was proposed in *JLS3* to let programmers concentrate on the logic inside the loop body and not worry about managing loop indexing. The data in Figure 4(c) also confirm that the introduction of the enhanced-for statement has changed how programmers select looping constructs.

Findings:

1. Syntactic rules exhibit nontrivial dependency. For example, 6% of rule combinations show strong dependency with $> 50\%$ probability.

2. Rule usage is contextual and helps identify potential syntactic sugars to simplify a language or guide syntactic level, instead of the usual lexical level, code completion and suggestion.

6. Applications

The main purpose of our study is to understand programming language usage from the perspective of syntactic rules. We have investigated characteristics of both independent and dependent rule usage and how rule usage evolves. Our results suggest several potential applications, which we discuss next.

Language design and restriction Programming language design has been largely artistic, driven by language architects' aesthetic concerns and intuitions. Typically language designers have limited knowledge on how programmers may actually use a language. As languages (such as Java and C++) may gradually introduce new features, they become more complex and impose additional obstacles for novices to learn. Recall that the *PSRP* values discussed in Section 3.1 show that *not all*, but a subset of syntactic rules is adopted in a single project. Inspired by the

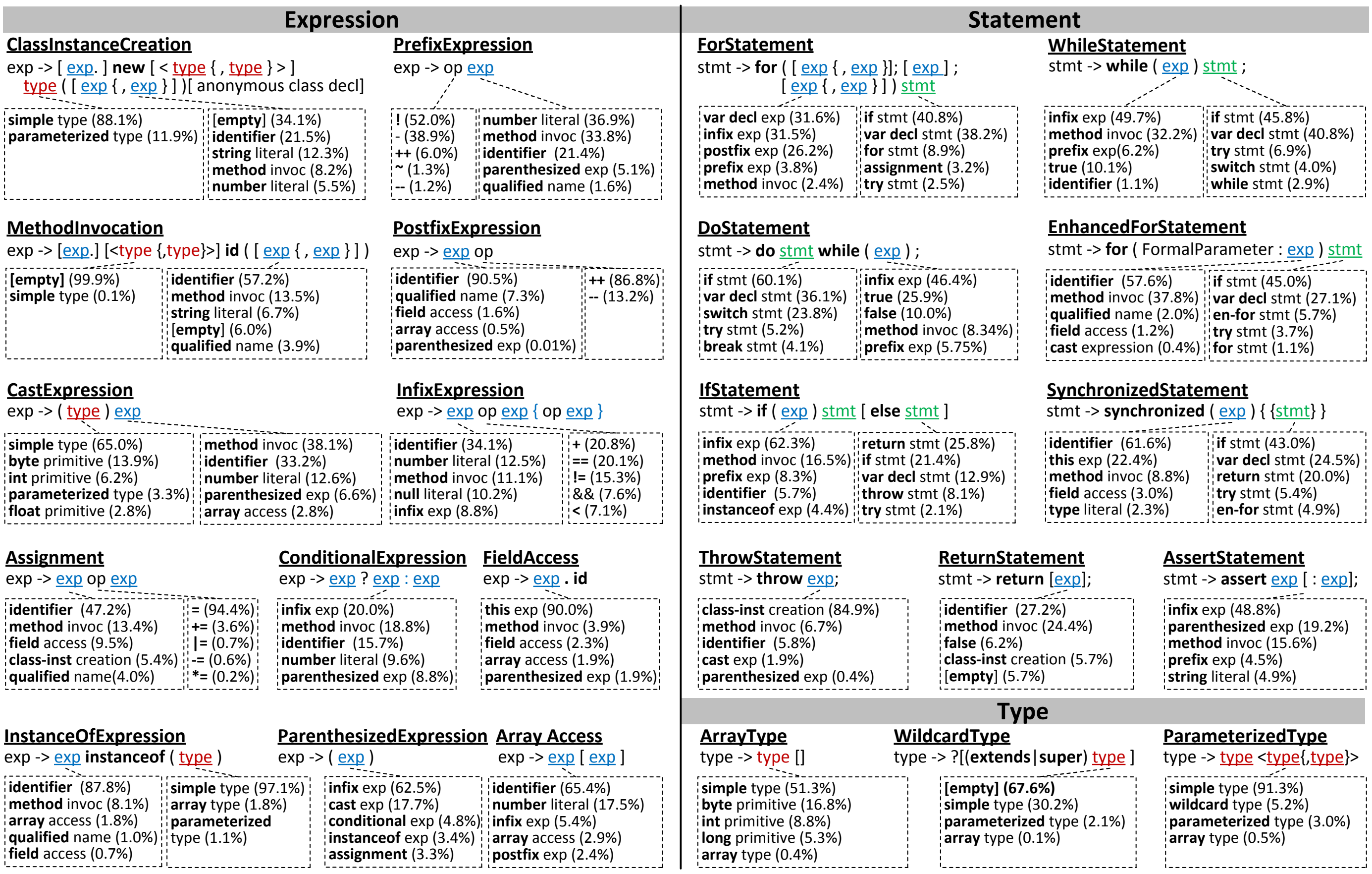

Figure 7: Cheat sheet of rule dependency use.

concept of compact profiles[6] defined in Java 8, we envision the construction of syntactic rule subsets from mined rule usage information. A programmer may adopt a proper subset for a given scenario. In addition, rarely used syntactic rules or ones with significantly decreased usage over time can guide language designers in optimizing/redesigning the rules. Programmers can also be warned so that they can use these rules more judiciously.

Programming languages contain not only *good* features, but also *bad* features. Poorly-designed features usually induce programmers to write bad code, which may further impact the quality of the software. In Java, for example, the `super-field` access expression permits customers to visit the fields of its super class directly. However, the use of this rule meanwhile violates the principle of information hiding. In this study, we found many bad practices in using the syntactic rules, *e.g.* ignoring caught exceptions (see Section 5.2, also check the list in Section 3.2). In order to prevent language customers from abusing these features, restricting the use of certain *bad* features is indispensable. By analyzing the usage of the syntactic rules, we can learn and construct this subset (without restricted rules) by given requirements (*e.g.* performance, reliability, security) from existing practical code and enforce employees/developers to use.

Identification of syntactic sugars Our results on depth-2 dependent rule usage have already motivated the automatic generation of syntactic sugars based on programmers' usage patterns. As discussed in Section 5.2, although the syntax of *for-each* has been applied in other programming languages, the use of the `for` statement in Java provides strong evidence that confirms the necessity of introducing the `enhanced-for` statement as a syntactic sugar. Our results also capture some *potential* syntactic sugars. For instance, the `if` statement always appear in the `else` part of another `if` statement, it would be significant to add a syntactic rule of *elseif*, like the elseif in PHP. In this study, we found 80 pairs of the rules that have strong use dependencies. It is interesting investigate and mine possible syntactic sugars. Moreover, deeper depth bounded dependent rule analysis may yield further opportunities for rule usage patterns.

Code recommendation and completion The strong contextual nature of syntactic rule usage promises a new potential code recommendation and completion technique based on structured syntax, rather than lexical tokens. As the depth-2 rule use dependencies have been calculated in Section 5, we adopt the conditional probability to predict the possible child rules that can be derived from the parent rule. Figure 8 shows an imaginary scenario that how syntax-based code recommendation is executed. When a user is entering a `while` statement, the system lists in-place candidate rules that user may adopt next, ranked by their contextual dependencies with the `while` statement. The user

[6] A *compact profile* is a subset of the full Java SE API, which has a smaller storage footprint and enables Java applications to run on resource-constrained devices [29].

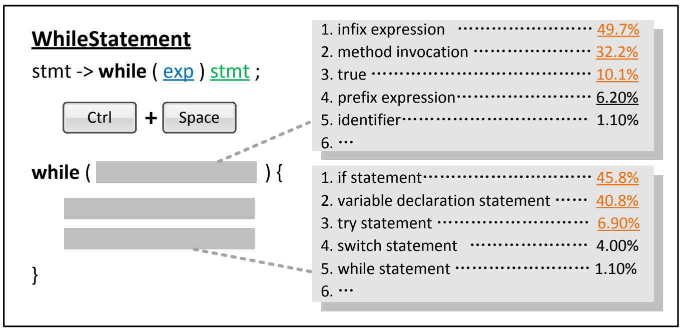
WhileStatement
stmt -> while (exp) stmt ;
Ctrl
+
Space
while (
) {
}
1. infix expression ········· 49.7%
2. method invocation ········· 32.2%
3. true ········· 10.1%
4. prefix expression········· 6.20%
5. identifier········· 1.10%
6. ···
1. if statement········· 45.8%
2. variable declaration statement ······ 40.8%
3. try statement ········· 6.90%
4. switch statement ········· 4.00%
5. while statement ········· 1.10%
6. ···

Figure 8: An example of syntax-based code recommendation. When a user types the keyword `while`, presses the a recommendation command (such as Ctrl+Space here), and the candidate rules are enumerated for both inner expression part and statement part in the `while` statement.

can select the appropriate rule and the recommendation process is iterative. In addition, we found, a parent rule derives, on average, only 20 syntactic rules, in which top 3∼5 of them dominate. That means, the user only need to lookup within a relatively small scope and consume less time and energy. We believe, this completion technique is lightweight; it only completes the skeleton of source code and leaves the concrete implementation to the programmers. It is also a worthwhile complement to current completion techniques.

7. Threats to Validity

Construct validity The construct validity of our study rests on the measurements performed, in particular related to the corpus construction, evolution analysis of rule usage.

Regarding the corpus construction, we select and download over $5,000$ projects with different characteristics, such as project size and domain. All projects are from GitHub, as it is easier to programmatically checkout multiple snapshots of the projects by Git. GitHub is one of the most popular project hosting service and hosts a great number of diverse projects. Thus, there is no indication that the projects we selected are biased toward any specific project types. In addition, many projects, which contain multiple languages in their implementations and are not dominated by Java are also included in our corpus. In some extreme cases, projects only involve 1 Java file. However, these are not toy projects since we selected them based on their popularity in Github. Hence, it is hard to filter and eradicate them on the premise of not including bias.

Regarding the evolution analysis of rule usage, to obtain the per year rule usage data, we automatically check out multiple versions from the code repository. By default, we select the first commit after a given time as a project's representative version for the year. However, not every project has a continuous development history. Some projects were interrupted by several months, even several years in some cases. This may cause the time of the snapshots that we checked out to be inconsistent with expectations. However, it is unavoidable to some extent. Based on our observations, only a few projects suffer from this issue, which does not affect the overall results.

External validity Threats to external validity are concerned with whether the results are applicable in general. The $5,646$ projects we study are all from open-source communities, our conclusions on syntactic rule usage may deviate from the results on commercial projects. In addition, all projects under analysis are implemented in Java; no other language is studied. However, the study process is general, which can be easily applied for other languages. For the future work, it would be desirable to analyze more kinds of projects (including commercial projects), developed in different programming languages to confirm our general conclusions.

8. Related Work

Relatively few studies empirically analyze language syntax usage in recent years. Fewer focus on the evolution of the syntax adoption and syntactic rule dependency usage.

Studies of Programming Languages. As with many other things, Knuth was one of the first to conduct an empirical study of how programmers use syntax on 440 Fortran programs [1]. He obtained the distribution of statement types through static analysis and project profiles through dynamic analysis. Based on the obtained knowledge on how Fortran was actually used by developers, he provided several strategies to optimize the compiler. Knuth's study has inspired additional researchers to consider how a programming language is used by developers. A variety of languages were studied, *e.g.* COBOL [6, 7], APL [8] and Pascal [9], through similar mechanisms. Our work can be viewed as a modern follow-up to Knuth's work to study a popular, mature and widely-used programming language. Besides the conventional analysis of independent rule usage, we further conducted more comprehensive studies, including the evolution of the rule usage and dependent rule usage.

Dyer *et al.* conducted a large-scale study on Java features usage [12, 30]. We have identified some of the same results, *e.g.* some of the features are most popular while several ones are rarely used by developers. In essence, however, our work differs in several ways: (i) their work mainly focused on the usage of the newly-introduced syntactic rules of Java's three newest editions, while we examined *all* syntactic rules, including the rules they studied; (ii) We studied rule usage evolution of *all* syntactic rules, while their work concentrated on the adoption of the newly-added rules by developers; and (iii) We studied dependent rule usage while theirs did not. Some other studies focused on a small set of selected language features, *e.g.* generics adoption in Java [31–33] and C++ [34], the reflection usage in Java [11]. Instead, we comprehensively studied language features from the perspective of language syntax.

Baxter *et al.* [35] presented the first in-depth study of the structure of Java programs through analyzing 56 projects. They measured the key structural attributes to check whether they follow power-laws. Grechanik *et al.* also mined structural usage in more than 2,000 Java projects [36]. In addition, Collberg *et al.* presented a study of the static structure of Java byte code programs [37]. They obtained both simple counts, *e.g.* methods per class, instructions per method and instructions per basic block, and complex structure metrics, *e.g.* the complexity of

CFGs. In contrast to their work, we formulated such structure attributes as syntactic rule usage dependencies, and conducted a more comprehensive analysis. Furthermore, we also analyzed usage of the newly introduced rules and their impact on existing rules, which they did not consider.

Kim and Yi [38] studied the usage of syntactic sugar in Java and C#. They focused on the `for` and `enhanced-for` statements, and analyzed their usage in 10 C# projects and 10 Java projects. They found that C# developers preferred to use syntactic sugar while Java developers use relatively less syntactic sugar. In our study, we analyzed the evolution of the usage of loop-related rules. We found that the usage of the `enhanced-for` statement has approached the usage of the `for` statement, and the introduction of the `enhanced-for` statement greatly reduced the usage of the other loop rules.

Meyerovich and Rabkin identified the factors that lead to programming language adoption [39], *e.g.* prior language skills, availability of open source tools. Ray *et al.* studied how programming language impacts the code quality from multiple dimensions [40]. They found language design did have a significant, but modest effect on software quality. Their work inspired an interesting question that how syntax impact the language adoption and the code quality.

Studies of Software Characteristics. As more open source repositories (*e.g.* Github, Sourceforge) have been publicly available, many researchers have started to learn software characteristics through empirical approaches. Gabel and Su studied software *uniqueness* [41]. They found that software generally lacks uniqueness which most code snippets we need to write already exist. Hindle *et al.* studied *naturalness* that actual code is "regular and predictable", like natural language utterances [42]. They followed the uniqueness study and confirmed the "syntactic redundancy" of software. Tu *et al.* further studied the *localness* that human-written programs were localized [43]. They introduced a cache language model that optimized the n-gram model by involving local regularities of the code to improve code suggestion accuracy. Our study studied software from the perspective of the language syntax, and found many syntactic rule usage is predictable. Allamanis and Sutton used nonparametric Bayesian probabilistic tree substitution to mine idioms from source code [44]. Their idioms are rules in a tree substitution grammar inferred from ASTs; our rule dependencies directly study the actual usage of the rules of a programming language's grammar; studying the relationship between these two views of ASTs is future work. Many other studies mined the vocabularies of the programming language to obtain the "word" usage statistics [45, 46]. Instead, we learned the usage distribution of the syntax rules.

Programming Language Education. Recent studies have shown evidences that syntax of a programming language remains a significant barrier to novice computer science students [5]. Denny *et al.* conducted a study to investigate the language syntax barrier for novice programmers [4]. They collected syntax errors from students' course exercises during a drill and practice activity. They found students often struggled with language syntax, even when writing short fragments of code. Stefik *et al.* also performed a controlled experiment to analyze what syntactical elements from programming languages have effects on the correctness of the novice programmer's use of language constructs [5]. Our study analyzed more programs in practice, exhibited the actual use of the language that can be applied by language designer to better optimize the design of the language. We followed the same objective to ease the barrier for language learners.

9. Conclusion and Future Work

We have presented a large-scale study of how Java's language syntax is used in practice using more than $5,000$ open-source Java projects. Our study has exposed interesting quantitative information to help understand how Java's syntactic rules have been used, both individually and considering contextual dependencies. This work enables and promotes a data-driven approach to language design.

There are several interesting directions for future work. First, we plan to conduct a more comprehensive study with other programming languages to increase the external validity of our findings. Second, we are interested in investigating the possibility of using rule dependencies to facilitate syntax-based code completion. Third, we plan to provide additional suggestions on improving language design, *e.g.* by constructing more easy-to-use syntactic sugar. Finally, we would like to understand how different language syntax features are used from more perspectives, *e.g.* does the number of developers in the project correlate with the number of rules used; does the project category affect the distribution of the rule usage; does the adoption of one or several specific rules improve the defect rate and reduce the code quality.

Acknowledgment

The work is supported by the National Natural Science Foundation of China under Grant No.61572126, the Huawei Innovation Research Program (HIRP) under Grant No.YB2013120195 and the Scientific Research Foundation of Graduation School of Southeast University Grant No.YBJJ1313.

References

[1] D. E. Knuth, An empirical study of fortran programs, Software: Practice and Experience 1 (2) (1971) 105–133.

[2] Strangest language feature, http://stackoverflow.com/questions/1995113/strangest-language-feature.

[3] Your language sucks, https://wiki.theory.org/YourLanguageSucks#Java_sucks_because.

[4] P. Denny, A. Luxton-Reilly, E. Tempero, J. Hendrickx, Understanding the syntax barrier for novices, in: Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2011, pp. 208–212.

[5] A. Stefik, S. Siebert, An empirical investigation into programming language syntax, Trans. Comput. Educ. 13 (4) (2013) 19:1–19:40.

[6] A. Salvadori, J. Gordon, C. Capstick, Static profile of COBOL programs, SIGPLAN Not. 10 (8) (1975) 20–33.

[7] R. J. Chevance, T. Heidet, Static profile and dynamic behavior of COBOL programs, SIGPLAN Not. 13 (4) (1978) 44–57.

[8] H. J. Saal, Z. Weiss, An empirical study of APL programs, Computer Languages 2 (3) (1977) 47–59.

[9] R. P. Cook, I. Lee, A contextual analysis of Pascal programs, Software: Practice and Experience 12 (2) (1982) 195–203.

[10] C. Parnin, C. Bird, E. Murphy-Hill, Java generics adoption: How new features are introduced, championed, or ignored, in: 8th Working Conference on Mining Software Repositories (MSR), 2011, pp. 3–12.

[11] B. Livshits, J. Whaley, M. S. Lam, Reflection analysis for Java, in: Asian Conference on Programming Languages and Systems (APLAS), 2005, pp. 139–160.

[12] R. Dyer, H. Rajan, H. A. Nguyen, T. N. Nguyen, Mining billions of AST nodes to study actual and potential usage of Java language features, in: International Conference on Software Engineering (ICSE), 2014, pp. 779–790.

[13] J. Gosling, B. Joy, G. L. Steele, The Java Language Specification, Addison-Wesley Longman Publishing Co., 1996.

[14] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Second Edition, Addison-Wesley Publishing Co., 2000.

[15] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Third Edition, Addison-Wesley Professional, 2005.

[16] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, A. Buckley, The Java Language Specification, Java SE 7 Edition, Addison-Wesley Professional, 2013.

[17] Eclipse EGit, http://www.eclipse.org/egit/.

[18] Eclipse JDT, http://www.eclipse.org/jdt/.

[19] Neo4j, http://www.neo4j.org.

[20] D. M. W. Powers, Applications and explanations of zipf's law, in: Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning, NeMLaP3/CoNLL '98, 1998, pp. 151–160.

[21] C. V. Lopes, J. Ossher, How scale affects structure in java programs, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015, pp. 675–694.

[22] J. Bloch, Effective Java (2nd Ed.), Prentice Hall, 2008.

[23] Compound Annual Growth Rate, http://en.wikipedia.org/wiki/Compound_annual_growth_rate.

[24] E. W. Dijkstra, Letters to the editor: Go to statement considered harmful, Commun. ACM 11 (3) (1968) 147–148.

[25] B. Eckel, Thinking in Java (4th Ed.), Prentice Hall, 2005.

[26] Java Labels, http://programmers.stackexchange.com/questions/185944/java-labels-to-be-or-not-to-be.

[27] Programming with assertions, http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html.

[28] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall, 2008.

[29] Compact profiles, https://docs.oracle.com/javase/8/docs/technotes/guides/compactprofiles/compactprofiles.html.

[30] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, Boa: A language and infrastructure for analyzing ultra-large-scale software repositories, in: International Conference on Software Engineering (ICSE), 2013, pp. 422–431.

[31] H. A. Basit, D. C. Rajapakse, S. Jarzabek, An empirical study on limits of clone unification using generics, in: International Conference on Software Engineering and Knowledge Engineering (SEKE), 2005, pp. 109–114.

[32] M. Hoppe, S. Hanenberg, Do developers benefit from generic types?: An empirical comparison of generic and raw types in Java, in: ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2013, pp. 457–474.

[33] C. Parnin, C. Bird, E. Murphy-Hill, Adoption and use of Java generics, Empirical Software Engineering 18 (6) (2013) 1047–1089.

[34] A. Sutton, R. Holeman, J. Maletic, Identification of idiom usage in C++ generic libraries, in: 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10, 2010, pp. 160–169.

[35] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, E. Tempero, Understanding the shape of Java software, in: ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA), 2006, pp. 397–412.

[36] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, C. Ghezzi, An empirical investigation into a large-scale Java open source code repository, in: ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2010, pp. 11:1–11:10.

[37] C. Collberg, G. Myles, M. Stepp, An empirical study of Java bytecode programs, Software: Practice and Experience 37 (6) (2007) 581–641.

[38] D. Kim, G. Yi, Measuring syntactic sugar usage in programming languages: An empirical study of C# and Java projects, Advances in Computer Science and its Applications Lecture Notes in Electrical Engineering 279 (2014) 279–284.

[39] L. A. Meyerovich, A. S. Rabkin, Empirical analysis of programming language adoption, in: ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2013, pp. 1–18.

[40] B. Ray, D. Posnett, V. Filkov, P. Devanbu, A large scale study of programming languages and code quality in github, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 155–165.

[41] M. Gabel, Z. Su, A study of the uniqueness of source code, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2010, pp. 147–156.

[42] A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, in: International Conference on Software Engineering (ICSE), 2012, pp. 837–847.

[43] Z. Tu, Z. Su, P. Devanbu, On the localness of software, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 269–280.

[44] M. Allamanis, C. Sutton, Mining idioms from source code, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 472–483.

[45] D. P. Delorey, C. D. Knutson, M. Davies, Mining programming language vocabularies from source code, in: 21st Conference of the Psychology of Programming Group (PPIG), 2009.

[46] E. Linstead, L. Hughes, C. Lopes, P. Baldi, Exploring Java software vocabulary: A search and mining perspective, in: Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation, 2009, pp. 29–32.