Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features

Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen
Iowa State University, USA
{rdyer,hridesh,hoan,tien}@iastate.edu

ABSTRACT

Programming languages evolve over time, adding additional language features to simplify common tasks and make the language easier to use. For example, the Java Language Specification has four editions and is currently drafting a fifth. While the addition of language features is driven by an assumed need by the community (often with direct requests for such features), there is little empirical evidence demonstrating how these new features are adopted by developers once released. In this paper, we analyze over 31k open-source Java projects representing over 9 million Java files, which when parsed contain over 18 billion AST nodes. We analyze this corpus to find uses of new Java language features over time. Our study gives interesting insights, such as: there are millions of places features could potentially be used but weren't; developers convert existing code to use new features; and we found thousands of instances of potential resource handling bugs.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages; Human Factors

Keywords

Java; empirical study; language feature use; software mining

1. INTRODUCTION

The Java Language Specification (JLS) [17–20] is the official specification for Java. New editions of the specification (JLS2–JLS4) are released as the language evolves to add new features. The official Java platforms (Java Runtime Environment (JRE) and Java Development Kit (JDK); Standard (SE), Mobile (ME), and Enterprise Editions (EE)) all implement the language based on this official specification.

Changes to the specification are driven by needs from the community. This need often comes in the form of an official request (a Java Specification Request (JSR)) using the Java Community Process (JCP). The JSR formally defines what the need is, why the current specification is lacking, and proposes a solution. Each new language feature has an accompanying JSR and each new edition of the language has an umbrella JSR to identify the new features.

Currently however, there is little quantitative evidence demonstrating how most of these new language features are used in practice. Previous studies have investigated the use of certain Java language features, e.g. [21] investigated the use of several object-oriented features in Java, such as class, interface, and method usage and [27] investigated the use of generics in Java. Similarly, [24], [9], and [10] investigated the use of non-language features such as reflection (which in Java, is supported by the runtime and not the language). However, these studies looked at a relatively small number of Java projects (around 20), investigated a very small subset of features, or did not investigate their adoption over time.

In this paper, we utilize the Boa language and infrastructure [14, 15, 28] to study Java feature adoption over time for 18 language features and on a large corpus of projects[1]. The dataset we query is over 31k projects from SourceForge [12], representing over 9 million unique Java source files, with over 28 million snapshots of those files, which when parsed contain over 18 billion AST nodes.

From this dataset we investigate if features were indeed anticipated by the community, by looking for their uses before their release dates. Our results show this is true: every feature is used prior to release. We then investigate how those features are adopted over time along three dimensions: *projects*, *source code files*, and *committers using the features*. Our results show that while some features are widely used, many see only limited use.

We then investigate if these features aren't being used due to lack of opportunity, by defining a set of mining tasks to locate source code that could potentially use these new features. We find millions of such cases, both in files existing before the feature's release date and in new files created after the feature's release. This suggests there is room for better tool support to recommend the use of these new language features or to convert code to use these features. It also suggests there may be a need for better training and advertisement of new features. Some of our interesting results include:

- All language features were used prior to their official release, indicating anticipation of such features.
- All studied features are used, however a few features are clearly the most popular, including: annotation use, enhanced-for loops, and variables with generic types. Several features saw minimal use.
- Developers do convert existing code to utilize new language features after their release. Thus, tool support for such con-

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2756-5/14/05...$15.00
http://dx.doi.org/10.1145/2568225.2568295

[1]Our queries and raw results are available online: `http://boa.cs.iastate.edu/java-features/`

version operations and recommendation of code locations to convert is important for the community.

- We found many instances where features could have been used, but were not, indicating a need for better training or IDE support. In fact, some missed opportunities could actually lead to erroneous behavior.
- Committers tend to adopt new features on an individual basis rather than in a team. This result is consistent with a previous study [27], but with 100 times more committers.
- Most committers use only a small number of new features. A small number of committers account for the majority of new language feature uses.

Next, we give background on each edition of the JLS and the new language features. Then in Section 3 we pose the research questions our study aims to answer. We describe the approach used in our study in Section 4 and give the study itself in Section 5. We give some discussion in Section 6. In Section 7 we describe some previous studies regarding language feature use. Finally we conclude with future work in Section 8.

2. JAVA LANGUAGE SPECIFICATIONS (JLS)

This section provides background on the Java Language Specification (JLS) [17–20]. Since the original edition of the Java Language Specification, there have been three updates. In this section we outline some of the changes to the language for each edition. The full list of features is shown in Figure 1. Note that new language features are purely additive - each edition is fully backwards compatible with previous editions.

2.1 JLS2 New Language Features

The Java Language Specification, edition 2 (JLS2) [18] was a relatively minor update in terms of new language features. This edition added one new language feature: *assert statements*.

2.2 JLS3 New Language Features

The Java Language Specification, edition 3 (JLS3) [19] added several significant language features, including: *annotation types*, *enhanced-for loops*, *type-safe enumerations* (enums), *generic types*, and *variable-argument methods* (varargs).

2.3 JLS4 New Language Features

The Java Language Specification, Java SE 7 edition (JLS4) [20] made several changes, including: *binary literals*, a *diamond operator* for generic type inference, *allowing catching multiple exception types*, *suppression of varargs warnings*, *automatic resource management*, and *underscores in literals*. As these features are not as widely known, we detail some of them in this section.

2.3.1 Type Inference for Generic Instance Creation (Diamond)

As previously mentioned, the language allows generic types. When declaring a variable of a generic type however, the generic type arguments must be repeated. For example:

```
Map<K, V> m = new HashMap<K,V>();
```

declares a `HashMap` with keys of type `K` and values of type `V`. Note that the generic type arguments were repeated in the variable declaration (left) and the object instantiation (right). JLS4 allows omitting the repeated generic type arguments in the instantiation (the so called *diamond operator*), thus changing the previous example to:

JLS2	
Assert	`assert i > 0;`
JLS3	
Annotation Declaration	`@interface Test { }`
Annotation Use	`@Test void m() { .. }`
Enhanced-For Loop	`for (T val : items) ..`
Enums	`enum E { N1, ..; }`
Generic Variable	`List<T> l;`
Generic Method	`<T> void m(T a) { .. }`
Generic Type	`interface List<T> { .. }`
Extends Wildcard	`Class<? extends E> c;`
Super Wildcard	`Class<? super S> c;`
Other Wildcard	`Class<?> c;`
Varargs	`void m(T... arg) { .. }`
JLS4	
Binary Literals	`int FIVE = 0b101;`
Diamond	`Map<K, V> m = new HashMap<>();`
MultiCatch	`catch (E1 | E2 e) { .. }`
Safe Varargs	`@SafeVarargs`
Try with Resources	`try (File f = new ..) { .. }`
Underscore Literals	`int MILLION = 1_000_000;`

Figure 1: Studied Java language features, with examples.

```
Map<K, V> m = new HashMap<>();
```

This new diamond operator can be used anywhere the compiler is able to infer the generic type arguments.

2.3.2 Catching Multiple Exception Types (Multi-Catch)

This edition allows specifying more than one exception type inside a catch clause. The catch clause's body is then executed when either exception type is caught. For example, the statement:

```
try { .. } catch (E1 | E2 e) { .. }
```

executes the catch statement's body if the try statement throws an exception of type `E1` or type `E2`. This helps avoid code duplication.

2.3.3 Safe Varargs Warning Suppression

The variable number of arguments in methods feature added in JLS2 can lead to a large number of compile-time warnings when combined with generics. Often however the programmer knows that these warnings can safely be ignored, so the ability to disable those warnings was added:

```
@SafeVarargs
@SuppressWarnings({"unchecked", "varargs"})
static <T> List<T> asList(T... elems) { .. }
```

The use of either of these annotations will suppress compiler warnings at this location.

2.3.4 Try with Resources

Certain resources, such as files, require manually releasing them when finished. This by itself is easy to forget, however even when programmers remember to close the resource, errors can still creep in [37]. To ease the management of these resources, a new statement was introduced:

```
try (File f = new ..) { .. }
```

This try statement declares a resource `f` which is available within the try statement's body. Upon exiting the try statement (either through normal or exceptional program flow) the resource is automatically released.

3. RESEARCH QUESTIONS REGARDING LANGUAGE FEATURE USE

The focus of our study is the usage of Java language features by open-source developers. In this section, we outline the specific research questions (RQ) we wish to answer.

RQ1: *Do projects use new language features before their release?* Often, especially with Java, an implementation of a requested feature is available before its release. This can take the form of an official beta/pre-release or an unofficial compiler.

We are interested in how often new language features are used prior to their official release. Such data can give an indication if a particular feature was anticipated and if providing implementations prior to release may be useful to the community.

RQ2: *How frequently is each language feature used?* The next question deals with feature usage. The addition of language features is driven by needs from the community, yet to date there has been no study to see how most of Java's language features are being adopted by developers.

This question examines language features introduced in JLS2–JLS4. For each language feature, its use across our entire dataset is tracked. This data gives insight into how each feature was, and is, being used.

RQ3: *How did committers adopt and use language features?* Once a new set of language features is available, it takes time for developers to learn how and where to use them. Some developers may be excited and try using them as often as possible. Other developers may be content with solving problems with the old set of features, as that is what they are accustomed to. We wish to investigate to see how language feature adoption occurs for individual developers.

RQ4: *Were there missed opportunities to use language features?* Although a new language feature may be available, developers might chose to not use it. We are interested in knowing how often such missed opportunities exist.

RQ5: *Was old code converted to use new language features?* We also wish to investigate to see if code using older language features is ever updated to use the newer language features.

4. APPROACH: TOOLS AND DATASET

In this section, we describe our approach for answering the previously identified research questions. Our approach relies on Boa [14, 15, 28] and its dataset from SourceForge [12].

4.1 Background on Boa Language and Infrastructure

The Boa language and infrastructure was designed to abstract away the details of software mining and provide a platform for easily writing queries that execute efficiently against a very large set of software repository data. Boa contains data from SourceForge projects and supports a wide range of queries on that data.

The Boa language abstracts away most of the details of software mining. The Boa framework mines the software repositories (in this case, SourceForge) and transforms the data into a custom set of types. The language provides these domain-specific types, such as `Project`, `CodeRepository`, and `Revision` that allow users to perform queries against software repositories.

Boa currently processes CVS and Subversion repositories of Java projects. When it finds a change to a Java source file, it checks out the snapshot of that file at that revision and parses it using the Eclipse JDT parser [2]. The parsed data is then translated into a custom representation, including types such as `Namespace`, `Declaration`, `Method`, `Variable`, `Statement`, and `Expression`. The statement and expression types are union types, allowing each to represent multiple cases.

The Boa infrastructure generates a Hadoop [5] MapReduce [11] program to efficiently execute queries. This is also abstracted away and developers do not have to explicitly write any code to parallelize their queries.

4.2 Dataset Used in Our Study

The dataset used for our study is the September 2013 dataset from Boa. This dataset includes all Java projects on SourceForge with at least one CVS or Subversion repository. The dataset does not include Java projects with only Mercurial, Git, or Bazaar repositories. The total number of Java projects is over 35k (see Figure 2).

Metric	Count
All Projects	*699,331*
Java Projects	*35,341*
Studied Projects	**31,432**
Repositories	*32,555*
Revisions	**9,557,448**
Files	*41,733,495*
File Snapshots	*86,411,272*
Java Files	**9,093,216**
Java File Snapshots	**28,747,948**
AST Nodes	**18,323,905,323**

Figure 2: Metrics for the SourceForge-based dataset in Boa.

However, not all of these projects are useful. We identified almost 4k projects where all Java source files contained a parse error. We filtered those projects out, leaving over 31k projects in the dataset for use in our study.

The dataset contains widely-used Java projects, including: Azureus/Vuze, Weka, Hibernate, JHotDraw, JabRef, JUnit, iText, FindBugs, JML, TightVNC, etc. This dataset represents over 9 million revisions by more than 50k developers. It contains over 9 million unique Java files and over 28 million snapshots of those files. This represents (to the best of our knowledge) the largest empirical dataset to date for Java projects that contains both full history information of the source repositories with over a decade of history and the full AST information from the Java source files.

For our research questions, the size of the Java projects (whether 1 or 1k files) is irrelevant, as we are interested in investigating Java language features used by developers without constraining the study to any specific kind of developer. Thus we include small projects (perhaps written by novice developers) as well as large projects (perhaps written by experts). However for RQ3, smaller projects could affect our results and thus as we mention later, for this research question, we filtered projects with few developers.

5. EMPIRICAL STUDY ON JAVA LANGUAGE FEATURE ADOPTION

This section presents our study on Java language feature usage.

5.1 RQ1: Do Projects Use New Language Features Before Their Release?

If a feature is requested by the community, then most likely people will be excited to use it prior to its release. To see if this is true, first we needed to know the release dates of official implementations for each language specification. We show these release dates, based on each specification's JSR, in Figure 3.

JLS2 (JSR 59) - Released 09 May 2002			
Feature	**Earliest Use**	**Projects**	**Files**
Assert	09 Feb 1998	114	1,068

JLS3 (JSR 176) - Released 30 Sep 2004			
Feature	**Earliest Use**	**Projects**	**Files**
Annotation Declaration	11 Nov 2003	7	130
Annotation Use	05 Jan 2002	12	1,165
Enhanced For	20 Jan 2002	44	634
Enums	05 Jan 2002	20	173
Generic Variable	11 Jul 1998	59	2,311
Generic Method	04 May 1999	22	919
Generic Type	01 Jul 1998	31	2,047
Extends Wildcard	02 Jan 2002	18	587
Super Wildcard	24 Jul 2003	3	426
Other Wildcard	10 Feb 2002	23	649
Varargs	23 Jul 2003	10	76

JLS4 (JSR 366) - Released 20 Jul 2011			
Feature	**Earliest Use**	**Projects**	**Files**
Binary Literals	04 Nov 2010	2	4
Diamond	01 Aug 2010	12	399
MultiCatch	01 Aug 2010	9	95
SafeVarargs	30 Apr 2011	3	17
Try with Resources	04 Nov 2010	8	109
Underscore Literal	04 Nov 2010	2	2

Figure 3: Language features are used before their release. (Note: cutoff times were midnight UTC on release date)

Using the release dates in this table, we then analyzed each valid Java file to see if it used a particular feature. We filtered out any Java file containing a parse error. Then we collected the timestamps of each file using each language feature and then filtered based on the particular language feature's release date. The results are shown in Figure 3 and include the list of features, the date of the first mined use of the feature, the number of projects that used the feature prior to its release, and the total number of files that used the feature prior to its release.

For the earliest uses, we manually investigated to verify the identified files actually used the particular feature and that the commit date matched our results. Based on this analysis we identified one project with clearly erroneous commit dates[2] and we removed that project from this analysis. Interestingly, for one project that made heavy use of generics in 1998, the commit log referenced "switch[ing] to GJ", which is the language extension proposed by Bracha *et al.* [8] that eventually became the basis of Java's generics.

The results in the table clearly show that every language feature was used prior to its official release date and adoption is most likely driven by compiler support. Next we investigate how each language feature was adopted over time.

5.2 RQ2: How Frequently Is Each Language Feature Used?

The addition of language features is driven by needs from the community. In this section, we quantitatively investigate how developers use these new features by looking at each unique Java source-file path in the system and taking the last existing snapshot of each. We then analyze that set of snapshots and count feature usage. For each file, we generate a mapping between features and the total uses in the file.

We show the results in Figure 4, first by total number of uses across the entire dataset, then by percentage of Java files using the feature, and finally by percentage of projects using the feature. The table clearly shows every feature is being used at least once. One trend that becomes readily apparent is that JLS4 features are not used very often, compared to JLS3 features. This is despite the fact there were over a million revisions and 3k Java projects active since the release of JLS4.

JLS2	**Assert**
Uses	408,802
File	1.04%
Project	12.72%

JLS3	**Annotation Declaration**	**Annotation Use**
Uses	29,415	11,692,911
File	0.28%	21.98%
Project	6.35%	57.07%

JLS3	**Enhanced For Loop**	**Enums**	**Generic Variable**	**Generic Method**
Uses	2,666,411	162,445	2,473,581	257,921
File	8.4%	1.47%	9.24%	0.9%
Project	48.61%	29.42%	57.15%	13.52%

JLS3	**Generic Type**	**Extends Wildcard**	**Super Wildcard**	**Other Wildcard**	**Varargs**
Uses	214,012	411,940	84,602	936,546	221,322
File	1.89%	1.37%	0.15%	2.45%	0.95%
Project	19.87%	15.67%	2.71%	22.52%	15.43%

JLS4	**Binary Literals**	**Diamond**	**MultiCatch**	**Safe Varargs**	**Try with Resources**	**Underscore Literals**
Uses	90	22,473	1,920	192	1,597	889
File	0%	0.08%	0.01%	0%	0.01%	0%
Project	0.02%	0.4%	0.27%	0.06%	0.21%	0.02%

Figure 4: Java language feature usage by total number of uses, by percent of all files, and by percent of all projects.

Also observe the trends for the ratios of uses to files. For example, the Annotation Declaration feature has a ratio close to one[3]; there is roughly one annotation declaration per file. This is similar for Enums and Generic Type. These features represent types in Java and thus one generally expects to see one type per file. The ratios for the other features are higher (2–6) since they are expressions and statements. For example, the ratio of enhanced-for loops is three[4] meaning files using the feature use it around three times.

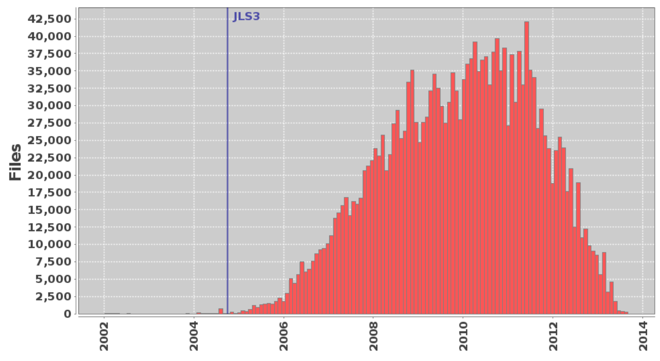
To see how features were adopted over time, we plotted histograms of each feature's use, both by number of files and by number of projects. After examining these plots for each feature, we noticed similar trends. They fell into two categories: JLS4 features and non-JLS4 features. Since the trends are similar across features, we picked representatives from each category.

The histograms contain bins with 30-day time ranges. The first time a feature appears, it is added to the respective bin. See Fig-

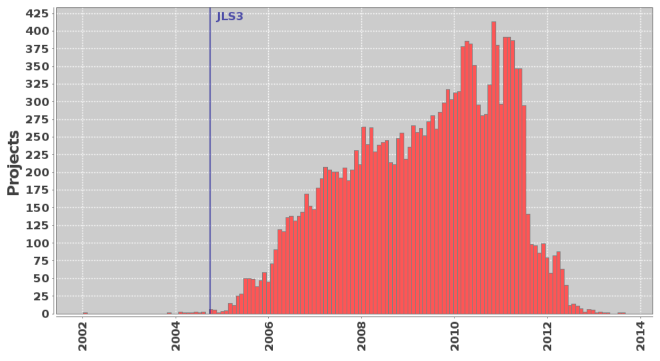
[2]http://goo.gl/l4f8S9

[3]This feature appears in 0.28% of files, or around 25k files, and is used around 29k times total.

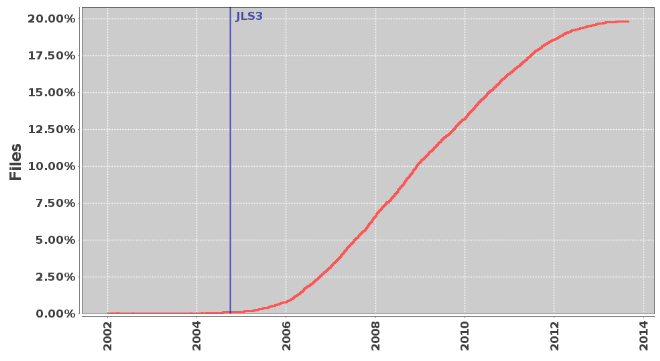
[4]This feature appears in 8.4% of files, or around 763k files, and is used around 2.6m times total.



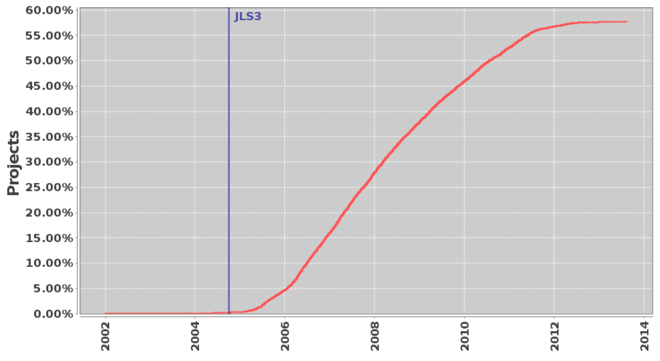
(a) First uses, by File



(b) First uses, by Project

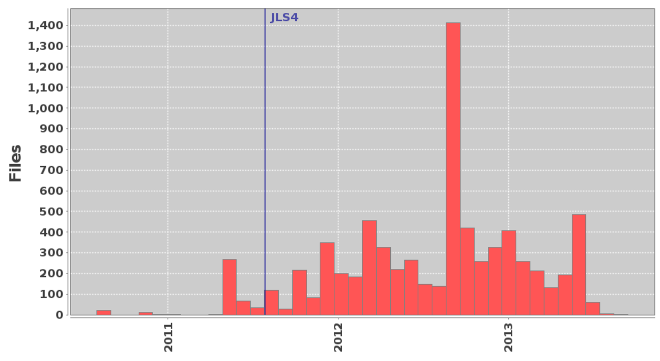


(c) Use Density, by File

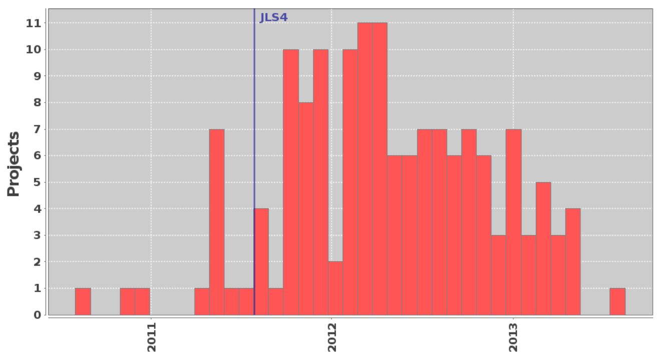


(d) Use Density, by Project

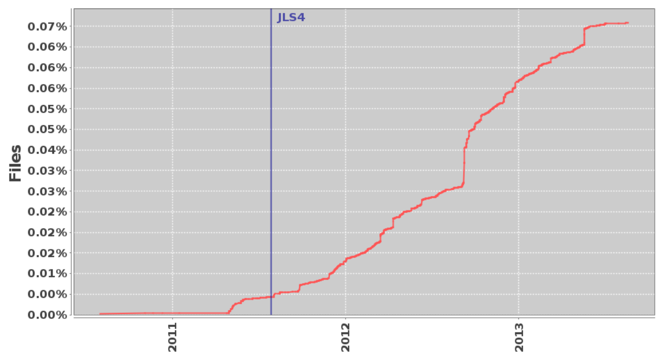
Figure 5: Use of the *Annotation Use* language feature.



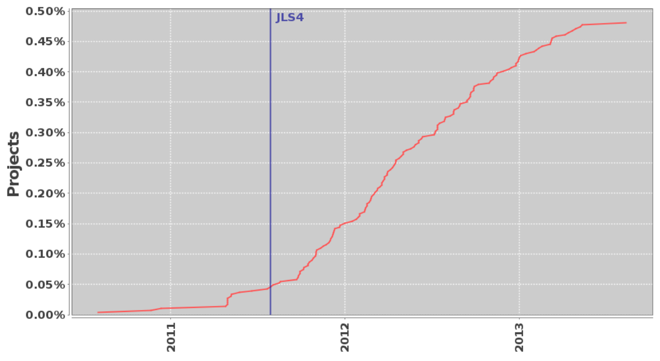
(a) First uses, by File



(b) First uses, by Project



(c) Use Density, by File



(d) Use Density, by Project

Figure 6: Use of the *Diamond* language feature.

ures 5a–5b and Figures 6a–6b. The plots also contain marker lines to indicate the release date of each JLS.

We also plotted densities of each feature's use, both by number of files and by number of projects. Points in these charts represent the number of files/projects using a feature at that time, divided by the total number of Java files/projects at that time, to account for growing repositories. See Figures 5c–5d and Figures 6c–6d.

For example, Figure 5 shows a non-JLS4 feature, Annotation Use. The histograms all show increasing adoption of the features after release with peaks around 2011. Then the number of files/projects adopting the feature for the first time starts decreasing. To better understand this decrease, we investigate the density plots.

As can be expected, the files and projects in the system were increasing over time. The density plots remove this variable from our analysis, by computing the percent of feature use at each time. For example, when we look at Figures 5c–5d we see that even as the total number of files and projects in the system increases, the relative percent is increasing too. Thus we can see that over time, the use of the feature is consistently increasing for all features studied.

Notice that Figure 6, a JLS4 feature, doesn't show as strong of trends as the previous two features discussed. In this chart, the histograms have less of an obvious trend to them, due to the relatively low number of total uses for this new feature. While the density graphs still show the same general trend of increasing use, both by files and by projects, there is less of a defined curve in these graphs.

Investigating Frequently Used Features.

As seen in Figure 4, most language features are used in a very small number of files (2% or less). The exceptions are Annotation Use, Enhanced For, and Generic Variable declarations. We further investigate some of these popular language features.

First let's look into the use of annotations, by collecting the annotation types named at each use. Figure 7 shows the top-ten frequently used annotation types and the number of uses for each. As can be seen, almost half of the annotation uses were the `@Override` annotation. Such widespread use of this annotation makes sense as IDEs such as Eclipse typically automatically add this annotation. This still counts as adoption however, as developers accept and commit these automatically added annotations. The second most used annotation, `@Test`, is used by unit testing frameworks. In fact, other than `@Test` and `@SubL`, the annotations listed are all JDK or J2EE provided annotations. We anticipated high use of JDK annotations, as the Annotation Declaration language feature has less than 0.3% use across all Java files, but the clear domination of those annotations was surprising.

Annotation Name	Uses	Percent
@Override	5,534,089	47.33%
@Test	981,737	8.40%
@SuppressWarnings	634,697	5.43%
@Column	246,467	2.11%
@XmlElement	140,754	1.20%
@SubL	134,990	1.15%
@Generated	131,759	1.13%
@XmlAttribute	101,156	0.87%
@XmlAccessorType	81,140	0.69%
@Deprecated	80,217	0.69%

Figure 7: Annotation uses. Percent is out of all annotation uses.

Next let's look into the generic variable declarations, by collecting the counts of each declared generic variable's type. Figure 8 shows the top-ten frequent generic types used (top) and the top-ten parameterized types (bottom). The results clearly show that the majority of generics are from collection types, the most common being `List<String>`. These results are consistent with the previously published study on generics use by Parnin *et al.* [27], although our study was on a thousand times more projects.

Generic Type	Uses	Percent
List	3,628,998	32.31%
ArrayList	2,145,612	19.10%
Map	1,156,480	10.30%
HashMap	842,934	7.50%
Set	811,990	7.23%
Collection	643,047	5.73%
Vector	570,016	5.07%
Class	547,628	4.88%
Iterator	500,887	4.46%
HashSet	384,408	3.42%

Generic Type	Uses	Percent
List<String>	514,339	22.68%
ArrayList<String>	416,306	18.35%
Class<?>	295,554	13.03%
Map<String, String>	208,195	9.18%
Map<String, Object>	177,048	7.81%
Set<String>	170,727	7.53%
HashMap<String, String>	148,861	6.56%
Vector<String>	137,706	6.07%
HashSet<String>	110,424	4.87%
HashMap<String, Object>	89,088	3.93%

Figure 8: Variables declared with generic types.

5.3 RQ3: How Did Committers Adopt and Use Language Features?

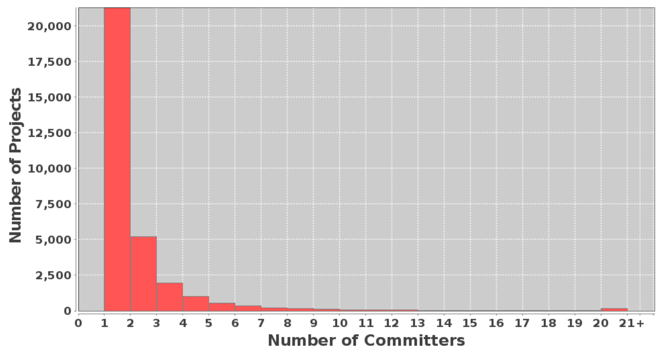
While in RQ1 we showed that all features are adopted before their release, and in RQ2 we showed how features are adopted over time, so far we have only evaluated feature adoption in terms of files and projects. In this section, we wish to evaluate if similar adoption trends also apply in terms of committers. Specifically, we also wish to study the adoption behavior of individual committers.

To do that, for each changed or added file that was recognized as containing a feature, we collected its commit time and author. For each commit that has changed files containing the use of a feature for the first time, the corresponding author is counted as one committer using that feature. The number of committed files containing the new features are also recorded and counted toward the number of uses for the corresponding committer. The threat to this method of counting is that if a committer uses a feature in a file which has already contained that feature (introduced by some other committer), they would not be counted. However, this threat is minimized because in this dataset a file is usually owned and edited by one or a few committers (as shown in Figure 9b).

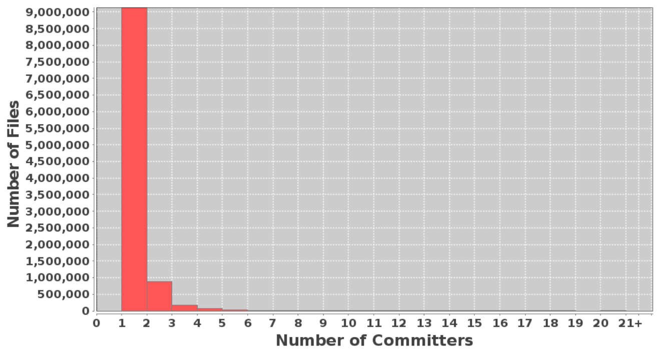
5.3.1 RQ3.1: How Many Committers Adopted and Used New Features over Time?

Figures 10 and 11 show the result for the number of committers using two different features over time. Each bar shows the number of users in the corresponding month. Even though the features appeared at different times, both show the same trend of adoption: a few committers used the feature before its release, then the number of users increases to a peak, and finally decreases. This is consistent with the adoption trend for projects and files seen in RQ2.

Among the committers using a feature, we counted the ones who used that feature for the first time (the lower area, in red) and the



(a) Number of committers in a project



(b) Number of committers editing a file

Figure 9: Number of committers per-project and per-file for Java projects in SourceForge.

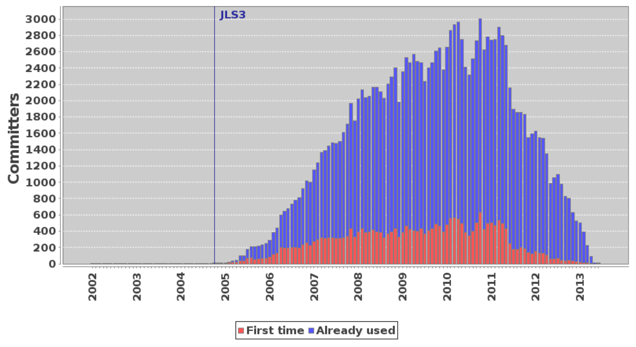


Figure 10: Committers use of Annotations over time.

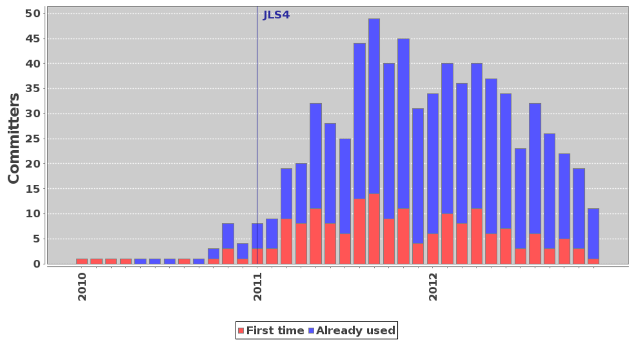


Figure 11: Committers use of Diamond over time.

ones who had used that feature before (the upper area, in blue). As seen in the figures, after the release date more committers adopted the new features. Once a feature is used for the first time, many committers kept using it in later commits (in blue). After a while, the number of first-time users (in red) decreases. This trend is the same for all the features in our study. Comparing the charts, the number of committers using Annotations is much higher than that for Diamond. This result is expected and is consistent with RQ2.

5.3.2 *RQ3.2: How Much Did Committers Use Each Feature?*

To answer this question, we count the number of uses of each feature for each committer. Since different features are used at different levels of granularity in the source code, e.g. generic fields can only be declared at the type level while enhanced-for loops can be used multiple times in the body of the method, we used the granularity of files to compute the number of uses. That is, the number of uses for a committer is the number of files to which that committer was the first one introducing that particular feature.

Figure 12 shows the result for two features: Annotation Use (12a) and Diamond (12b). In each chart, the x-axis represents the committers ranked by their number of uses and the y-axis (in logarithmic scale) is the number of uses. Each bar represents the number of uses for a single committer. Notice that a small number of committers accounts for a large number of feature uses. About half of the number of committers introduced a feature to less than 10 files, while a few committers used the feature in tens, hundreds, to thousands files. This trend holds for all features.

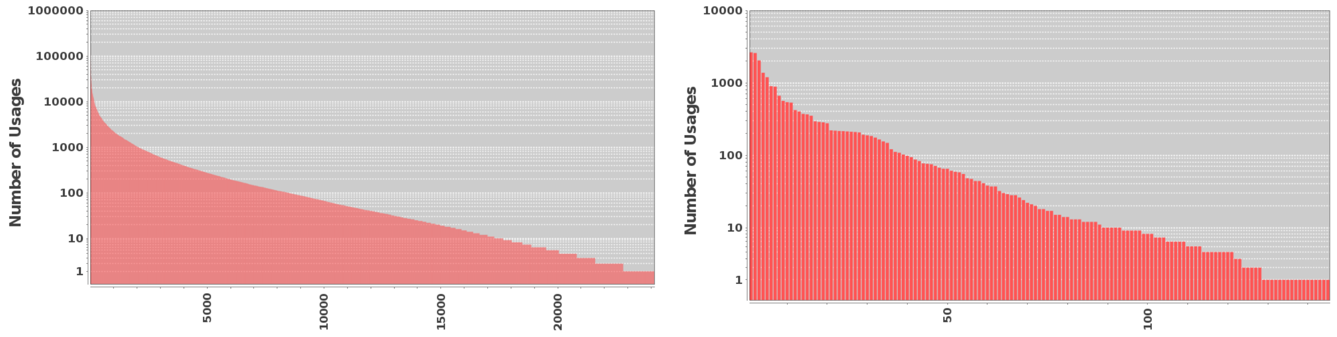
Comparing the charts, we can see that the number of committers are quite different: about 24,000 for Annotation Use (12a) and about 150 for Diamond (12b). In addition, the number of committers with the same number of uses varies among features. For example, at 10 uses, there are about 17,000 and 90 committers, respectively. This suggests that there are some feature(s) which are more popular and widely-used (e.g. Annotation).

5.3.3 *RQ3.3: Did Committers Adopt Features on an Individual Basis or As a Team?*

To answer this question, we investigated how many team members adopted features in each project. We first collected the set of committers for each project, identified how many times each committer used a feature, and ranked the committers per-project based on their number of uses. Then, for the top-k committers (for k=1,2,3), we computed the proportion of the top-k committers's uses over the total number of uses in the whole project.

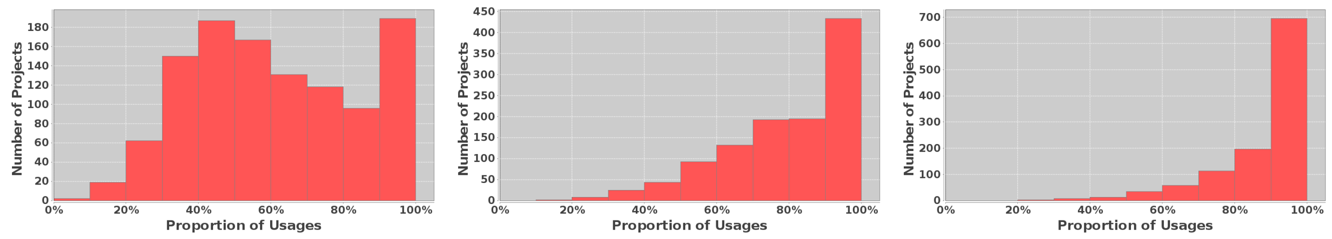
In Figure 9a, we can see that the distribution of the number of committers in a project is right-skewed. That is, many projects have only a few committers. In those projects, only one or two committers contribute almost 100% of the uses. To avoid that bias and to study the team culture, we filtered out all projects having less than six committers [34]. After filtering there were 1,429 projects remaining, which we used for this study.

The result is shown in Figure 13. Each chart shows the histogram of the proportion of feature usage in projects. The bins are the ranges 1-10%, 11-20%, ..., and 91-100%. Figure 13a shows the result for the top-1 user. A single committer contributes 100% of uses in over 150 projects and 90% of uses in almost 300 projects. In Figure 13c, when considering the top-3 users, the number of 90%



(a) Annotation Use (b) Diamond

Figure 12: Use of language features by committers.



(a) Top-1 Annotation Use committer (b) Top-2 Annotation Use committers (c) Top-3 Annotation Use committers

Figure 13: Proportion of feature uses in projects.

uses increases to almost 900 projects, which are the majority of the 1,121 projects that use that feature. The other features (not shown) follow similar trends.

This result indicates that a feature is not widely adopted by all members of the team, but instead are mainly championed by a small number of members. This is also consistent with a previous study [27] even though they studied only 20 projects while we studied 1,429 projects (with at least six committers each).

5.3.4 *RQ3.4: Did Committers Use All Features?*

For this question, we track the feature uses of a group of "active" committers, who routinely committed code over a long enough period. Since JLS3 had the most new features, we used the set of committers at the release time of JLS3. We kept all committers that had routinely committed code at least every 6 months in the time between releases of JLS3 and JLS4. Filtering for committers that used at least one language feature in our study, the remaining set contained 61 committers. The scatter graph in Figure 14 shows their uses over time. For better visualization, we group related features from the same edition, i.e., Annotation Declaration/Use into Annotation, generics features into Generics, Binary/Underscore Literals into Literal, and Try with Resources and MultiCatch into TryCatch. A horizontal line shows the use over time for a single committer.

As seen from the graph, among the 61 committers only committer #24 adopted features from all three editions. Most committers used features from JLS2 and JLS3. JLS4 was only used by committers 42 and 24. Most of the committers used Assert, the only new feature in JLS2, however, they started late after its release. Meanwhile, the committers adopted JLS3 quite early and most of them used several different features. In terms of individual feature uses, up to now, no committer has used all studied features. Committers have used at most 7 out of 10 different grouped features.

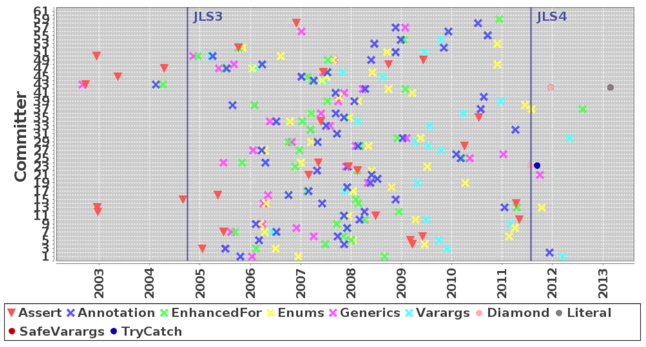


Figure 14: Tracking features used by committers.

5.4 RQ4: Were There Missed Opportunities to Use Language Features?

In this section we investigate missed opportunities to use new language features, by mining the latest snapshot of source code to find locations where new language features could potentially be used. For example, we mined to find integer literals with 7 or more characters that did not use underscores. We also mined methods that have as their first statement an if condition that if true throws an `IllegalArgumentException` (which could potentially be turned into an assert statement), methods that take an array as last argument instead of a varargs argument, expressions where the literal '1' was shifted left (which could use binary literals), generic instantiations that don't use the diamond pattern, try statements with more than one catch block having the same body, and try statements with a call to a `close()` method in the finally block.

The results are shown in Figure 15. In the first row, we list the number of mined potential uses in files that existed prior to the feature's release. These represent places where a maintainer could convert code to use the new language feature. We found tens of thousands (to millions) of potential uses in old files.

	Assert	Varargs	Binary Literals	Diamond	MultiCatch	Try with Resources	Underscore Literals
Old	89K	612K	56K	3.3M	341K	489K	22.2M
New	291K	1.6M	5K	414K	24K	33K	2.3M
All	380K	2.2M	61K	3.7M	365K	522K	24.5M
Files	1.39%	12.74%	0.11%	12.25%	2.28%	1.85%	20.17%
Projects	18.18%	88.78%	5.9%	59.08%	49.75%	37.27%	88.86%

Figure 15: Potential language feature uses, in old files (before feature release) and new files (after feature release).

The second line of the table shows potential uses in files that were added after the release of the feature. These are locations that developers had the option to use a language feature, but did not. Again, we found thousands of potential uses for each feature and even millions of potential uses for two features.

While some of this unused potential has small impact, such as underscore and binary literals making code more readable, other missed opportunities could actually lead to erroneous behavior. Specifically, we investigate regarding the try with resources language feature which aims to properly close resources. As Weimer and Necula [37] point out, this is a common source of bugs in programs. For example, the code:

```
BufferedReader br = ...;
String s = br.readLine();
br.close();
```

wouldn't call `close` if the call to `readLine` throws an exception.

While we found over 500k potential uses for this language feature, we were interested in how many of those might lead to buggy behavior. We narrowed the results of the algorithm to only include methods that throw `IOException`, do not catch that exception anywhere in the body, and contain a call to a `close()` method. We found 193,768 instances of potential[5] resource handling bugs!

5.5 RQ5: Was Old Code Converted to Use New Language Features?

As we showed in the last section, when new language features are released there is potentially a lot of existing code that could have used the new feature. In this section we investigate if developers convert old code to update it to the new language features.

Unlike the last section where we used only the latest snapshot, in this analysis we mine each version of a file and compute the number of potential and actual uses of a language feature. We then compute those values on the previous version of the file. If the number of potential uses decreases by **exactly** the amount the actual uses increased, we consider it a potential conversion. This analysis is extremely conservative and may miss a lot of conversions, but it should give a low number of false positives and make verification easier, and allows us to confirm the existence of conversion activities to use new features. We show the results in Figure 16.

[5]Manual verification of 30 random samples showed 50% accuracy.

	Assert	Varargs	Diamond	MultiCatch	Try with Resources	Underscore Literals
Count	180	2.1K	8.5K	162	154	2
Files	105	1.6K	3.8K	125	99	1
Projects	37	488	72	23	17	1

Figure 16: Detected conversions to use new features.

We verified the results by manually checking (up to) 30 detected files from each feature, grouping the results by project and systematically sampling from a random starting point in the list. When verifying a file, if other files from the **same revision** were in the dataset we also verified those. In total we verified 2,598 out of 5,694 files as direct conversions, 13 as not conversions, and 4 as more complex conversions that also added the new feature.

During this process we found several commit logs mentioning conversions to JDK7 or specifically for one feature (e.g. Diamond, MultiCatch, Assert). One even stated "reviewing locations where 'throw' appears and substituting by 'assert' when convenient."[6]

As we showed, developers do convert existing code to utilize new language features after their release. Thus, tool support for converting operations to use new language features and recommendation of code locations to convert is important for developers.

5.6 Threats

We identified a threat regarding who commits code versus who actually wrote it. Someone may commit a file they did not write, perhaps adding a file from another library so it is local in their own repository. Our analysis would attribute the source of that file to the person who committed it which is why we focused on committers, not developers. Similarly, multiple committers may actually be the same person but count separately in our analysis.

A similar threat relates to the timestamps of committed code. If someone commits a file they did not write, the timestamp of the commit may be wrong. It is possible that features were actually used earlier than identified in RQ1.

We identified an external threat to our study regarding the generalizability of our results. Since we only studied open-source software, the results may not necessarily represent Java language feature usage by non-open source developers, such as those in industry. We also do not know the experience level of committers, which may vary greatly and limits our ability to generalize. We avoid generalizing our results and instead focus on if the trends we observed are similar to the trends the previous study [27] observed.

6. DISCUSSION

While we do not know *why* in general people seem to avoid using new language features, we did see a lot of unrealized potential to use these features. It may be the case there should be more or better training of developers, or perhaps better advertisement of new features. Or it may simply be user/project preference or the feature isn't viewed as useful by the broader community.

What was clear from our study however was two facts. First, there is indeed a lot of code that **could** use these new features, but currently does not. This seems to indicate a need for recommendation systems [13, 30, 33] to suggest using the newer language features. The second fact was that people do tend to convert code to use these newer language features. Conversions and recommendation systems go hand in hand.

[6]`http://goo.gl/5pyR0T`

For example, if a user wrote "`int i = 3000000;`" an IDE could show a suggestion to convert this code to use underscores, for better readability. Similarly if a literal value of 1 is shifted left, it could recommend using binary literal notation.

Currently, Netbeans [3] has an Inspect and Transform [4] feature that converts to use diamond, underscore literals, try with resources, and multicatch. Eclipse [1] will show a warning if you don't use the diamond operator (this behavior is disabled by default[7]) and provides a quick fix to remove the redundant type arguments. They also provide content assist for features such as Diamond and MultiCatch.

There is also room for improvement in IDEs. Not all features are enabled out of the box, some features may be slightly confusing (such as Eclipse giving a quick fix of 'surround with try/catch' and a second choice of 'surround with try/multicatch'), and not all features have conversions or recommendations.

7. RELATED STUDIES

Grechanik *et al.* [21] performed a large-scale study on Java features on 2k projects from SourceForge. They provided a relational database and studied features such as: classes (abstract, nested, etc), methods (arities, return types, etc), fields, conditional statements, etc. The majority of the features studied are object-oriented language features available since JLS1. They did not study newer language features in JLS3 or JLS4. Their study also focused on releases of projects and not the full history of the repositories.

Parnin *et al.* [27] mined the history of 20 open-source Java projects to evaluate how Java generics were integrated and adopted into open source software. As we already showed, our finding on the most popular generic types is consistent with their empirical result. It is also true for the finding that generics are usually adopted by individuals championing for the features, rather than all committers in the team. Hoppe and Hanenberg [22] performed a small empirical study to determine if generic types in Java provide benefit to developers. Basit *et al.* [7] performed an empirical study on two projects regarding how Java generics and C++ templates can help in code refactoring.

Livshits *et al.* [24] focused on the reflection feature in Java. They introduced a static-analysis based reflection resolution algorithm that uses points-to analysis to approximate the targets of reflective calls as part of the call graph. Callaú *et al.* [9] studied the reflection feature in Smalltalk. They reported that such a feature is mostly used in specific kinds of projects: core system libraries, development tools, and tests, rather than in regular applications.

Richards *et al.* [31] performed a large-scale study on the use of `eval` in JavaScript applications. `eval` is used to transform text into executable code, allowing programmers the ability to dynamically extend applications. They studied large-scale execution traces with 550k calls to the `eval` function exercised in over 10k websites. They found that it is often misused and many uses were unnecessary and could be replaced with equivalent and safer code. Earlier, Richards *et al.* [32] analyzed a smaller set of JavaScript programs and concluded the popular usage of `eval` and reported the degree of dynamism in those programs. Ratanaworabhan *et al.* [29] reported on an existing benchmark for JavaScript and focused on two aspects of JavaScript runtime behavior 1) functions and code and 2) events and handlers. Yue and Wang [38] performed an empirical study on almost 7k websites regarding insecure practices of JavaScript inclusion and dynamic generation. They reported that over 40% of the websites dangerously use `eval`.

Gorschek *et al.* [16] performed a large-scale study on how developers use object-oriented concepts. Tempero [35] studied how fields are used in Java and reported that it is common for developers to declare non-private fields, but then not take advantage of that access. Tempero *et al.* [36] found higher use of inheritance than expected and variation in the use of inheritance between interfaces and classes. Muschevici *et al.* [26] studied multiple dispatch in several languages and compared its uses.

Meyerovich and Rabkin [25] studied how programming languages are adopted by users, via several large surveys. Their study was focused on which languages were adopted and did not go into detail of specific language features.

The Sourcerer project [23] provides a relational database of mined software artifacts. Their dataset contains over 18k Java projects from SourceForge and Apache. The data is modeled as entities, such as classes, methods, or fields, and relationships among those entities. The dataset contains the source code from the latest snapshot of each project. Baldi *et al.* use the Sourcerer project and topic modeling to empirically validate the theory that aspects are latent topics with a high scattering entropy [6].

While these previous studies have looked at various language features, most are limited to studying a few features, looked at a relatively small number of projects, or did not look at the full history of the software studied. Our study looks at most of Java's new language features, studies over 31k Java projects, and uses each file's full history.

While Grechanik *et al.* [21] and Sourcerer provided potential datasets to use in this study, we chose to use the Boa infrastructure [14, 15, 28] with over 31k Java projects due to having full access to it as well as our intimiate familiarity with the infrastructure.

8. FUTURE WORK AND CONCLUSION

Programming languages evolve over time to meet the needs of developers. What was needed was a study to see how those features are actually used by developers. In this paper we investigated language feature usage for Java's three newest editions.

Our analysis revealed that some developers were eager to use these features, even using them as far as six years before their release. Our results showed that every feature is indeed used over time. The most-used features we studied were the enhanced-for loops, declaring variables of generic type, and using pre-defined annotations. The first two features are related and our analysis indicated their heavy use is influenced by the Collections classes provided by Java's runtime. The heavy use of annotations, but relative lack of custom annotations, indicated the use was mostly by automated tools such as IDEs or code generators.

Most features saw limited use, but our further investigation showed millions of additional places features could have been used but were not. This included old files that could be converted, as well as new code. We detected thousands of potential bugs in code that could be fixed by converting to use new features. We also detected many places where developers had actually converted existing code to use new features, indicating a need for tools to suggest and convert code to use new features.

In the future it would be interesting to perform a survey of the developers that appeared in this study to see why they chose to start using features when they did. Perhaps there is a need for better education/outreach to inform developers of these new features.

9. ACKNOWLEDGMENTS

This work was supported by National Science Foundations (NSF) grants CCF-13-49153, CCF-13-20578, TWC-12-23828, CCF-11-17937, CCF-10-17334, and CCF-10-18600.

[7]`http://goo.gl/EDvzOl`

10. REFERENCES

[1] Eclipse. http://www.eclipse.org/, 2014.

[2] Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/overview.php, 2014.

[3] Netbeans. http://www.netbeans.org/, 2014.

[4] Netbeans inspect and transform. https://netbeans.org/kb/docs/java/editor-inspect-transform.html#convert, 2014.

[5] Apache Software Foundation. Hadoop: Open source implementation of MapReduce. http://hadoop.apache.org/, 2014.

[6] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA, pages 543–562, 2008.

[7] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, SEKE, pages 109–114, 2005.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Not.*, 33(10), Oct. 1998.

[9] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: the case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR, pages 23–32, 2011.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th international conference on Static Analysis*, SAS, pages 1–18, 2003.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI, 2004.

[12] Dice Holdings, Inc. Sourceforge website. http://sourceforge.net/, 2014.

[13] E. Duala-Ekoko and M. P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP, pages 79–104, 2011.

[14] R. Dyer, H. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*, ICSE, pages 422–431, 2013.

[15] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, 2013.

[16] T. Gorschek, E. Tempero, and L. Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE, pages 115–124, 2010.

[17] J. Gosling, B. Joy, and G. Steele. *Java(TM) Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1996.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2000.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[20] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *Java(TM) Language Specification*. Prentice Hall, Java SE 7 edition, 2013.

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

[22] M. Hoppe and S. Hanenberg. Do developers benefit from generic types? An empirical comparison of generic and raw types in Java. In *4th ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH, 2013.

[23] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18, 2009.

[24] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS, pages 139–160, 2005.

[25] L. Meyerovich and A. Rabkin. Empirical analysis of programming language adoption. In *4th ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH, 2013.

[26] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA, pages 563–582, 2008.

[27] C. Parnin, C. Bird, and E. R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *8th IEEE International Working Conference on Mining Software Repositories*, MSR, 2011.

[28] H. Rajan, T. N. Nguyen, R. Dyer, and H. A. Nguyen. Boa website. http://boa.cs.iastate.edu/, 2014.

[29] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps, 2010.

[30] P. Resnick and H. R. Varian. Recommender systems. *Commun. ACM*, 40(3):56–58, 1997.

[31] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP, pages 52–78, 2011.

[32] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2010.

[33] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[34] S. R. Schach. *Object-oriented and Classical Software Engineering*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2005.

[35] E. Tempero. How fields are used in Java: An empirical study. In *Proceedings of the 20th Australian Software Engineering Conference*, ASWEC, pages 91–100, 2009.

[36] E. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP, pages 667–691, 2008.

[37] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA, pages 419–431, 2004.

[38] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th international conference on World Wide Web*, WWW, pages 961–970, 2009.