Analysis and Exploitation of Natural Software Diversity: The Case of API Usages

Diego Mendez, Benoit Baudry, Martin Monperrus

Analysis and Exploitation of Natural Software Diversity: The Case of API Usages

Diego Mendez, Benoit Baudry, Martin Monperrus
University of Lille & Inria

***Abstract*—In this paper, we study how object-oriented classes are used across thousands of software packages. We concentrate on "usage diversity", defined as the different statically observable combinations of methods called on the same object. We present empirical evidence that there is a significant usage diversity for many classes. For instance, we observe in our dataset that Java's String is used in 2 460 manners. Beyond those empirical observations, we show that we can use this API usage diversity to reason on the core design of object-oriented classes. We think that our pieces of evidence on API usage diversity shake up some established ideas on the nature of software and how to engineer it. Hence, we discuss those empirical results in the general context of software engineering: what are the reasons behind this diversity? what are the implications of this diversity?**

I. Introduction

Gabel and Su [10] have published fascinating results, showing that most pieces of code of less than 35 tokens are redundant. They appear elsewhere in the same project, or, for small sequences, elsewhere in the space of all ever-written software. In ecology, a sister concept of redundancy is *diversity*. In ecosystems, species are said to be redundant if they have the same functional role, and are said to be diverse if many different species occupy different niches.

There are many kinds of diversity in software [9]. In this paper, we focus on one kind of diversity: the usage diversity of classes of object-oriented code. Our main research question reads as follows.

> Do all developers use a given class in the same way? or in diverse ways?'

By "usage diversity", we mean ways of using a class in terms of method calls. We consider software from the viewpoint of *type-usages*, an abstraction introduced in [18], [19]. This concept abstracts over tokens, control flow and variables interplay. In a nutshell, a type-usage is a set of method calls done on a variable, parameter or field in a code base. For instance, Figure 1 presents a method body and three corresponding type-usages.

From a dataset of hundreds of thousands of Java classes, we have extracted millions of type-usages and measured their diversity (as defined by the number of different type-usages that can be observed). For instance, we have found that the Java class "String" is used in 2 460 different ways. This is not an exception, our experiment provides us with empirical evidence that a large scale diversity exists in "API usage"[1] of certain object-oriented classes.

We then provide original results on how to exploit the diversity of API usage in an actionable way. We demonstrate that the diverse usages of a given class capture valuable information about the number of responsibilities of that class. We also point how the API usage diversity can be analyzed to compare the expected usage by the class designer and the actual usage.

Our contributions are:

- a set of new software metrics, inspired by biodiversity metrics, that quantify the amount and the structure of diversity of API usage;
- the empirical observation of diversity of API usage in a large dataset;
- the exploitation of API diversity to reason on the design of object-oriented classes;
- a discussion of those results in the general context of software engineering: what are the reasons behind this diversity? what are the implications of this diversity?

If the literature includes a large amount of work on the synthesis of artificial diversity in software systems [9], to our opinion, our work is the first study that empirically quantifies the presence of diversity in object-oriented API usage. Hence, our work can be classified as ecology-inspired software engineering research [2], [21].

We think that our pieces of evidence on API usage diversity shake up some established ideas on the nature of software and how to engineer it. Some of our points are of speculative nature, but they aim at fostering a collaborative research effort on understanding the factors behind this API usage diversity. This paper is an extension of conference paper published at the 2013 International Working Conference on Source Code Analysis and Manipulation [16]: the new section VI discusses how we can use the topology of type-usages to reason on a class' semantics; section VII now clearly differentiates between reasons and implications of API diversity.

The rest of the paper reads as follows. Section II gives some background on object-orientation and type-usages. Section III describes our experimental design. Section IV exposes our empirical results and findings, while section V investigates

[1] We use the term "API usage" to reuse the same term as close work [12]. In this case, "API" refers to "Application Programming Interface", which at the level of a class, is defined by the set of exposed methods (whether "exposed" means public, documented of callable).

Source Code:

```
void saveNames(String inputPath) {
  ArrayList filenames = new ArrayList();
  File inputFile = new File(inputPath);
  if (inputFile.isDirectory()) {
    for (File f : inputFile.listFiles()) {
      filenames.add(f.getName()); }
  }
}
```

Abstraction:

```
// type usage #1 corresponds to "inputFile"
type:File
calls:{Constructor(String), isDirectory(), listFiles()}

// type usage #2  corresponds to "filenames"
type:ArrayList
calls:{Constructor(), add(String)}

// type usage #3  corresponds to "f"
type:File
calls:{getName()}
```

Fig. 1. Illustration of the concept of "types-usage". An extractor transforms the Java source code at the left hand-side into the abstraction at the right hand-side. Type-usages abstract over tokens, control flow and variables interplay.

potential biases in these observations. Section VI analyzes the diversity of usages with respect to the number of responsibilities of a class and the essentiality of methods in an API. Section VII discusses possible reasons for this diversity as well as possible implications on software engineering practices. Finally, related work (Section VIII) and conclusion (Section IX) close the paper.

II. Background

A. Object-oriented software

In object-oriented software, a class defines a set of functions (called methods) meant to be used in conjunction, in order to perform computations in a certain problem domain. For instance, in the problem domain of manipulating character strings, the Java class `String` defines 76 methods to use and transform strings in a variety of manners. The term "object" refers to an instance of a class.

In object-oriented software, variables can point to objects, and one "calls" methods on variables. Syntactically, this is written with a dot. Calling method "getFirstLetter" on a string variable is written `a.getFirstLetter()`. The method operates on the data that is encapsulated within the object. Designing the scope of methods and where to put them is all the art of object-oriented design.

B. Type-Usages

We consider software from the viewpoint of *type-usages*, an abstraction introduced in [18], [19]. A type-usage is an unordered set of method calls on the same variable of a given type occurring somewhere within the context of a particular class [19]. Type-usages abstract over tokens, control flow and variables interplay. Calls must be made on the same variable (whether local variable, method parameter or field), are unordered (the location in source code is not taken into account) and unique (observing several times the same call on the same variable is not taken into account). A call consists of the signature of the method to be called, that is, in Java, the method name, the parameter types (the methods *void init(String)* and *void init(File)* are considered as two different calls), and the return type. There is no distinction between instance methods and class methods ("static" in Java). A constructor call resulting in an object assigned to a variable is considered as a method call on this variable.

Example type-usages are shown in Figure 1. The left-hand side contains a piece of Java source code. The right-hand side lists the corresponding type-usages. For instance, type-usage #1 corresponds to variable *inputFile* which refers to an object created by a constructor call, on which two methods are called: "isDirectory" and "listFiles".

We say that type-usages are of the same "kind" when they have the same declared type the same set of calls. In the following, when we use "type-usage", we mean this aggregated set of identical items. To refer to a concrete type-usage (say, the one corresponding to variable "inputFile" in Figure 1), we will use the term "type-usage instance" (programming terminology) or "type-usage specimen" (ecology terminology). Along this line of thought, a type-usage corresponds to a species (as opposed to type-usage instances which are individuals).

III. Experimental Design

Our experiment consists of collecting a large number of type-usages across open-source Java applications and computing the corresponding values of novel bio-inspired software metrics.

A. Dataset

We have collected all Jar files present on a machine used for performing software mining experiments for 7 years. A Jar file is an archive containing compiled Java code under the form of a collection of ".class" files. We remove some duplicate Jar files with a heuristics based on file names. The resulting dataset contains 3 418 Jar files. Some Jars are still duplicated (the same version or very close versions) but this is no threat for the diversity measurement since the duplication does not introduce new type-usages. The residual duplication may still have an impact on the abundance of type-usages. The dataset only contains real code (mostly open-source code, but also binary proprietary code and student project code) and no artificial code that may have arisen along software mining. It represents 11 GB of Java bytecode and refers to 382 774 different types (classes or interfaces). The list of Jar files is given in the companion web page [15] and the raw

Abundance	
$abundance_{project}(typeusage)$	is the number of type-usages instances of a given type-usage for a single project (in $[0, \infty[$).
$abundance_{ecosystem}(typeusage)$	is the number of type-usages instances of a given type-usage in the ecosystem (in $[0, \infty[$).
$abundance_{project}(class)$	is the sum of all type-usage instances that are typed by the same class in a given project ($\sum abundance_{project}(typeusage)$, in $[0, \infty[$.
$abundance_{ecosystem}(class)$	is the sum of all type-usage instances that are typed by the same class in the ecosystem ($\sum abundance_{project}(class)$, in $[0, \infty[$.
Diversity	
$diversity_{project}(class)$	is the number of different type-usages of a given class for a single project (in $[0, \infty[$).
$diversity_{ecosystem}(class)$	is the number of different type-usages of a given class in the whole ecosystem (in $[0, \infty[$).

TABLE I
ECOLOGY-INSPIRED DIVERSITY METRICS FOR TYPES-USAGES.

data is available upon request. In this paper, for the ecological metaphor, we call this dataset the "ecosystem" under study.

B. Extraction Software

The extraction software extracts the type-usages described in II-B from Java code. It uses the analysis library Soot [24]. It works at the method body scope for local variables and method parameters and class scope for method calls done on fields. The extractor takes as input either Java source code or Java bytecode. It is publicly available on Github[2].

C. Metrics

The extraction of type-usages on our dataset yielded 9 022 262 type-usage specimen. We post-processed those type-usages to compute the metrics described in Table I. There are two groups of metrics: *"abundance metrics"* and *"diversity metrics"*. Metrics have two dimensions: 1) whether they are computed at the type-usage or class level 2) whether the are computed for a single project or for the whole dataset.

Those metrics are inspired from ecology. The abundance of species is the number of specimens (individuals), we define the abundance at the level of type-usages and classes. The abundance of a type-usage is the number of times it is observed in a given scope, i.e. the number of type-usage instances.

The richness of an ecosystem is one measure of diversity, it is the absolute number of species that can be observed in this ecosystem. In our context, the richness of an object-oriented class is the absolute number of different type-usages found in a given domain. We call this metric $diversity_{ecosystem}(class)$. A more precise definition is given in table I.

IV. EVIDENCE OF API USAGE DIVERSITY

For us, a very intriguing research question is: what is the diversity of usages of object-oriented APIs?

In other terms, do all developers use a given class in the same way? More formally, what are the values of $diversity_{ecosystem}$ as defined in table I? For us, a class would be "diverse" if we observe many different type-usages of this type in the ecosystem under study.

A. Abundance and Diversity Distribution

Figure 2 shows the distribution of the abundance and diversity at the level of classes in the ecosystem as boxplots

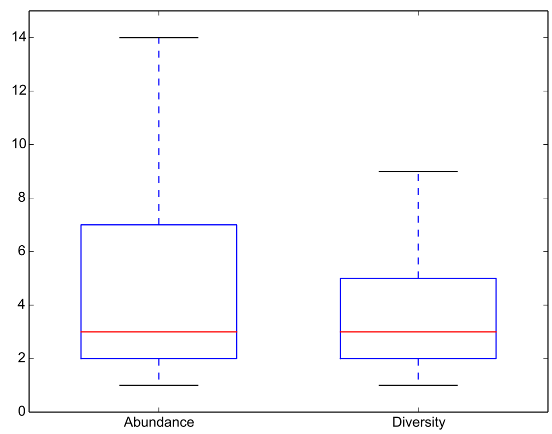

Fig. 2. The Type-usage Abundance and Diversity of All Classes of the Dataset Under Study. The outliers are not represented for sake of scale.

($abundance_{ecosystem}(class)$ and $diversity_{ecosystem}(class)$ of Table I). The median abundance is 3 (an abundance of 3 means that we have collected 3 type-usage instances for this class). The abundance boxplot shows that across the 382 774 classes of our dataset, a large majority are used a small number of times. This is due to the fact that many classes are only used in a single project (Jar file) of the dataset and within this project at most a handful of times.

The boxplot representing the distribution of diversity (second boxplot starting from left) shows that classes have a median number of 3 type-usages[3]. The upper quartile is 5. In other terms, for 75% of the classes, we observe between 1 and 5 ways of using of the class. *However, the data contains many extreme points that are not represented on the boxplot since their order of magnitude dwarfes this low diversity.*

B. Classes with High Usage Diversity

Let us now concentrate on the upper quartile of the diversity metric, those classes with high usage diversity. In our dataset, there are 748 classes for which we observe more than 100 different type-usages and 48 classes for which we observe

[2]https://github.com/monperrus/typeusage

[3]Note that the maximum diversity of a class is necessarily its abundance in the case where each type-usage specimen is different. It thus makes sense that the median diversity is 3 given a median abundance of 4.

Class Name	Diversity	# Methods	$\|TU\|=1$	$\|TU\|=2$	$\|TU\|=3$	$\|TU\|=4$	$\|TU\|=5$	$\|TU\|=6$	$\|TU\|=7$	$\|TU\|>7$
java.lang.String	2460	69	69	529	638	614	396	145	51	18
java.io.File	2166	47	45	373	775	613	264	69	17	10
java.lang.StringBuffer	1312	51	41	142	238	316	290	176	83	26
java.util.ArrayList	1236	36	36	179	307	328	236	115	29	6
java.lang.Class	872	62	62	333	286	115	45	18	8	5
java.util.List	724	31	30	149	235	194	86	23	5	2
java.lang.StringBuilder	643	44	42	92	139	142	132	63	22	11
org.eclipse.swt.widgets.Composite	639	227	135	222	131	86	41	16	4	4
javax.swing.JButton	625	143	83	119	141	102	72	43	21	44
javax.swing.JLabel	570	101	76	145	153	108	47	16	13	12
org.w3c.dom.Element	534	60	60	198	165	76	19	9	4	3
javax.swing.JPanel	530	108	77	115	116	112	65	31	12	2
org.w3c.dom.Node	516	39	38	128	150	95	46	29	18	12
java.util.HashMap	471	22	20	92	125	123	74	30	6	1
org.eclipse.core.resources.IFile	456	68	59	167	120	55	30	12	6	7
java.util.HashSet	453	23	23	75	134	120	77	19	5	0
org.eclipse.core.runtime.IPath	360	36	34	148	114	43	14	4	2	1
org.eclipse.swt.widgets.Label	312	97	56	83	68	68	23	8	3	3
javax.swing.JScrollPane	308	105	73	77	77	45	18	11	4	3
org.eclipse.swt.widgets.Display	247	157	108	86	34	9	4	1	2	3
org.w3c.dom.Document	209	61	56	79	45	17	6	3	2	1
org.eclipse.core.runtime.Path	192	48	25	61	62	33	7	4	0	0
org.eclipse.emf.common.util.EList	128	29	29	50	31	10	4	3	1	0
org.eclipse.core.runtime.IConfigurationElement	119	21	20	31	46	16	5	1	0	0
org.osgi.framework.Bundle	115	33	33	55	22	4	1	0	0	0
org.eclipse.core.runtime.IStatus	100	13	12	25	31	17	7	5	3	0
org.xml.sax.XMLReader	100	15	15	20	20	21	13	6	4	1
org.w3c.dom.Attr	94	22	19	33	22	14	3	1	1	1
org.eclipse.core.resources.IWorkspaceRoot	88	37	37	39	7	5	0	0	0	0
java.lang.Object	31	10	10	16	5	0	0	0	0	0

TABLE II

THE DIVERSITY OF 30 WIDELY USED API CLASSES AND THEIR NUMBER OF TYPE-USAGES PER SIZE IN NUMBER OF METHOD CALLS. THE COLUMNS $|TU| = n$ GIVE THE NUMBER OF TYPE-USAGES CONSISTING OF n METHOD CALLS (E.G.; THERE ARE 69 TYPE-USAGES OF ONE SINGLE METHOD CALLS FOR JAVA'S STRING).

more than 500 type-usages. The extreme case is Java's String. For this class, we observe 2 460 type-usages (among 394 959 type-usages specimen – instances – of type "String").

Table II gives the diversity of 30 diverse classes. The first column is $diversity_{ecosystem}(class)$ as defined in III-C. The second column is the number of called methods in the dataset. The columns $|TU| = n$ give the number of type-usages consisting of n method calls (e.g.; there are 69 type-usages of one single method calls for Java's String). Those 30 classes come from the following stratified sampling: the 10 most used classes of the Java Development Kit (JDK) in number of projects, the 10 most used classes of Eclipse (an important sub-ecosystem of our ecosystem) and the 10 most used classes that are neither from Eclipse nor from the JDK. We refer to the latter as "non-JDK classes", we show them to show that usage diversity does not only appear in JDK classes. For instance, there are 534 different type-usages for W3's "Element" and 639 for Eclipse's Composite.

As programmers, we were really surprised by this richness. Why were we surprised? Probably because of the implicit principle of software engineering stating that an abstraction (whether function, class or method) should do one single thing (coined the "Single Responsibility Principle" by Robert Martin [14]). In the perspective of type-usages, this principle reads as: 1) a class should have a small number of methods; 2) all methods should be used in the same way with some small variations. However, in our opinion, having hundreds of type-usages for certain classes is not a small variation.

Let us first deepen our understanding of this diversity before exploring the factors behind it.

C. Type-usage Dominance

Certain object-oriented classes give birth to a large diversity of type-usages. Now we would like to understand the structure of this diversity: are there type-usage that are much more used than the others?

Let us assume that we observe 1000 type-usage instances spread over 100 different type-usages. If 800 of them are of the same type-usage, that would mean that the type-usage diversity is actually *dominated* by a single one. To characterize this phenomenon, we define the *dominance* metric (called dom) as follows:

$freq_{ecosystem}(typeusage)$ is the frequency of a type-usage in the dataset (in $[0, 1]$).

$$= \frac{abundance_{ecosystem}(typeusage)}{\sum_i abundance_{ecosystem}(typeusage_i)}$$

$dom_{ecosystem}(class)$ is the maximum observed frequency among type-usages referring to the same class (in $[0, 1]$).

$$dom_{ecosystem}(class) = max(\{freq_i\}|type(i) = class\})$$

We have computed the type-usage dominance of the 382 774 classes of our dataset. Figure 3 gives the distribution as an

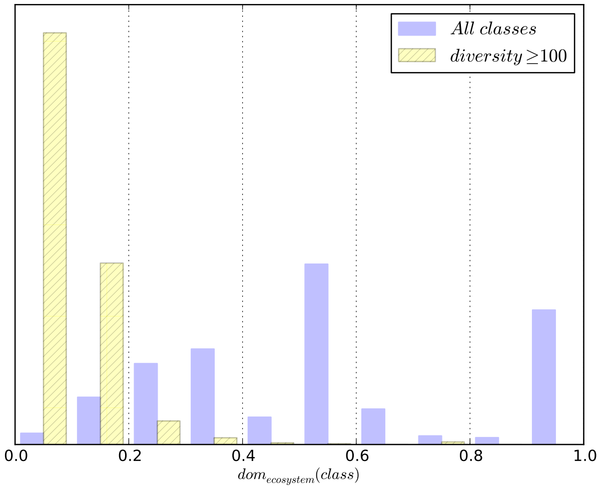

Fig. 3. The Distribution of Dominance as an Histogram, for all classes of the ecosystem and for very diverse ones. Diverse classes have no dominant type-usages.

histogram (the plain, unhatched bars). We observe two peaks around 0.5 and around 1. A dominance of 1 means that all type-usage specimens of a given class correspond to the same type-usage, i.e. that there is no diversity at all. A dominance of 0.5 means that half of the type-usage specimens are identical. Both cases are peculiarities of our dataset, corresponding to classes for which we observe one or two type-usage specimen. The rest of the distribution contains "dominated" classes ($dom > 0.5$) as well as classes for which there is no observed dominant type-usages (low dominance value, e.g. $dom < 0.3$). The latter correspond to classes where there is a real API usage diversity: nonetheless there are many type-usages but all of them are used in equal proportion. Now, let us come back to to the high diversity observed for certain classes.

Let us concentrate on those 748 classes for which we have observed more than 100 different type-usages. Are those classes really diverse? Java's String has a dominance of 0.083, the most frequent type-usage is indeed not dominant. Does this hold for the other very diverse classes as well? The hatched bars of Figure 3 give the dominance distribution of those 750 very diverse classes. Most classes have type-usage dominance lower than 0.2. The largest bin (the tallest hatched bar) corresponds to a dominance in the interval $[0, 0.1]$. For those classes, there is no "standard way" of using the class and the type-usage diversity does not correspond to "exotic variations".

To further demonstrate this point, Figure 4 plots the diversity and dominance values for each class of the ecosystem. The X axis is the diversity metric, the Y axis is the dominance metric. Each dot is a class. We can clearly see that there is a correlation between diversity and dominance: the more diversity, the less dominance. This confirms the findings on the 748 most diverse classes. Those pieces of evidence converge to state that *the API usage diversity we have observed previously is actually a true diversity*.

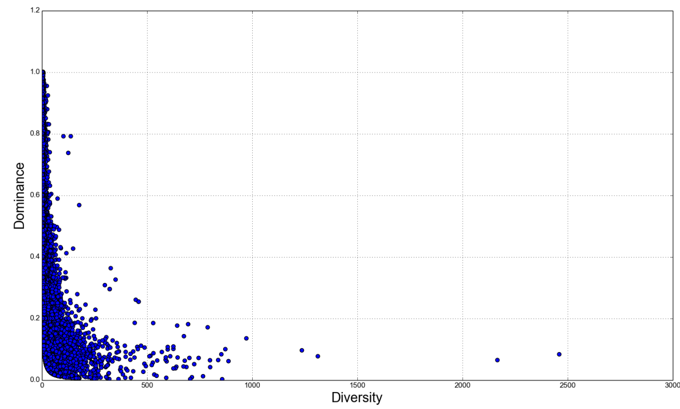

Fig. 4. Correlation between Diversity and Dominance. Each point of the graphic is a class. The more diverse a class' type-usage, the less dominance.

D. Usage Entropy of Classes

The dominance metric reflects the skewness of the distribution of the abundance of type-usages. However, it neglects the distribution of the rest of the distribution, the 2^{nd} most abundant type-usage, the 3^{rd}, etc. To compute the overall skewness, we propose to use Shannon's entropy. This enables us to deepen our answer to the research question on type-usage dominance.

In ecology, Shannon's entropy is an established diversity metric [11] ("diversity index" in the ecological terminology). In our context, the entropy formula for type-usages, which we call *u-entropy*, reads as follows:

$$\textit{u-entropy}(class) = -\sum freq(i) ln_2(freq(i))$$

where the i are all observed type-usages of a class and $freq$ is an abbreviation of $freq_{ecosystem}(typeusage)$. The entropy is correlated to diversity: the more entropy, the more diversity.

The entropy is maximum when all type-usages are equally distributed (i.e. of equal importance, with no dominance at all). In this case, $maxentropy(class) = -ln_2(diversity_{ecosystem}(class))$. This value is the theoretical maximum of the entropy, i.e. the maximum level of diversity. For all classes of the ecosystem, let us draw $maxentropy(class)$ versus $entropy(class)$, in order to see whether the maximum diversity is often approached or not.

Figure 5 is a scatter plot of the $\textit{u-entropy}(class)$ (X axis on a logarithmic scale) versus $ln_2(diversity(class))$ (Y axis), i.e. the maximum theoretical entropy. Those axes represent the two components of what ecologists call "species evenness". One dot is a class among the 382 774 classes of the ecosystem. The diagonal lines emerging from the points correspond to the theoretical maximum entropy (when the type-usages are uniformly distributed). There are no point for which $entropy(class) > ln_2(diversity(class))$ for obvious theoretical reasons. The vertical lines at the left-hand side of the figure correspond to all classes with a small number of type-usages (one line is $ln(diversity = 3)$, one line is

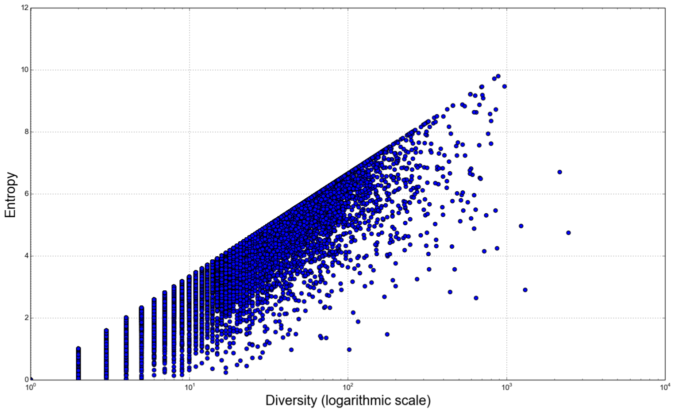

Fig. 5. The Type-Usage Entropy of Classes (Y axis) as a function of the API Usage Diversity (X axis). Each point of the graphic is a class. Most classes are grouped just below the maximum entropy, i.e. the diversity is almost systematic.

$ln(diversity = 4)$, etc). The main striking point of this figure is that *the cloud of points sticks to the maximum entropy*.

First, it further validates the finding of Figure 4. While the dominance only takes into account the most frequent type-usages, the entropy reflects the skewness of the whole distribution. Since the points are grouped along the maximum entropy, with no gap between, this also shows there is a tendency to real diversity (the type-usages are all used frequently). We would rephrase it as *the API usage diversity is systematic*.

Second, let us concentrate on classes which have the same diversity value (according to metric $diversity$ of Table I). This corresponds to a vertical line of points. We see that those lines can be quite high, especially for low values of $diversity$. This means that there is a kind of a "meta-diversity": the distribution of type-usage abundance does not follow a simple rule for all classes.

V. Discussion

We have reported in Section IV that there exists classes with very diverse API usages. This has never been observed before. Before going further in explaining and exploiting this diversity, let us dwell on the threats to the construct validity, i.e., on the threats that our measurement actually reflects the reality we claim to observe. In other terms, the research question we ask is: what is the reasonableness of our results?

A. An Artifact of the Extraction Software?

When we observed this phenomenon that has never been reported before, the first thing we did was to check our extraction software. We carefully browsed the list of type-usages for classes Map and String to check whether 1) they make sense, 2) they actually appear in code. The answer was positive. More generally, during our experiments, for six months, we browsed many extracted type-usages and the corresponding source code and this gives us confidence in our results.

B. Type-usages Result From Combinations of Method Calls

One reason behind this diversity is that type-usages are combinations of public methods. The second column of Table II is the number of externally used methods on instances of those classes (in-class and inherited methods). One sees that all diverse classes have a large number of methods, and that most methods appear in atomic type-usage with a single method call (e.g. for String, there are 69 used methods and 69 type-usages of size 1). To check whether the usage diversity only depends on the number of methods for very diverse classes, we compute the the Spearman correlation between the usage diversity and the number of public methods. The Spearman correlation is based on the ranks hence is independent of the exponential combinations of methods. On the 748 classes, the Spearman correlation is 0.25, which is low. The Spearman correlation is composed of numerical comparisons of the ranks of all pairs of classes. A low value of 0.25 means that there are many pairs of diverse classes whose diversity and number of methods go in opposite directions. indeed there are 40% of class pairs for which the diversity goes in opposite directions (less methods but greater diversity). This shows that the usage diversity is driven by more factors than only the number of public methods.

C. Objects are Used across Different Methods

Our analysis statically creates type-usages for local variables, method parameters and fields. If at runtime, an object is passed from methods to other ones, our analysis would output several type-usages, while at the runtime object level, all method calls would be done on the same object. For instance, let us consider a developer who wants to create a list, add elements and print them if the list is not empty. For some reasons, this developer would initialize the list in the class constructor, declare a new method for adding elements and at last, define a method that prints the elements and also checks that the list is not empty. As a result, we would have 3 different type-usages: <init>, <add>, <isEmpty, get>. We call those type-usages "type-usage fragments". However, at the object level, all method calls are done on the same object and the type-usage would be: <init, add, isEmpty, get>. In the extreme case, if 10 methods are called in ten different methods, we would produce 10 type-usages, while there would be actually one. In such case, our diversity measures would be artificially 10x too big.

To explore this hypothesis, we propose to study the size of type usages of a given class. The idea is that if we only have very small type-usages, our static analysis has probably only captured small, non atomic type-usage fragments. On the contrary, if there are large type-usages, the analysis is able to capture real interactions between methods on the same variable.

Table II presents the distribution of type-usages per type-usage size for the 30 reference classes. Recall that the columns $|TU| = n$ give the number of type-usages consisting of n method calls. Hence, the left-hand side columns contain small type-usages which are likely to be fragments. For instance,

Class	#cc	Explanation
Java Collection	2	One connected component (cc) is related to iterating over the elements a collection, the other one is about modifying the collection (adding elements).
Java Set	2	The same as Collection. This indeed makes sense because Set is a subtype of Collection in Java. This shows that the type-usage lattice reflects the inheritance of contracts.
Java Properties	2	One cc is related to getting properties ($getProperty$), the other one to creating properties. Interestingly, the intercession cc clearly contains the 4 main methods for creating property files: load, setProperty, put, putAll).
Java Class	2	One cc is related to class reflection, the other to array reflection. (In Java, an array is a class, but a special one. In particular, the component type of the array is accessible via a non regular, array-specific reflection method).
Java Matcher	2	One cc is related to testing the presence of patterns ("match" method), the other one to finding concrete occurrences ("find" method). This corresponds to 2 out of 3 documented responsibilities[4] of the class. The missing official responsibility ("lookingAt") is much less used in practice and consequently does not appear, given our filtering.
Java Thread	2	One cc is related to starting new threads, the other one is related to manipulating the class loader. Indeed, they are both actual, really different, responsibilities of Java's "Thread".
Java String	2	Both connected components are related to manipulating the string ("substring", "indexOf", etc.). One is structured around "substring", the other cc around "endsWith". This is not meaningful, it is an artifact of this particular threshold.
W3C Element	3	A class for representing XML nodes. Two connected components are about reading capabilities using methods for instance method and "getLocalName", "getAttribute"), the other one is about writing capabilities with "setAttribute" and "appendChild".

TABLE III

THE VALIDATION OF USING THE TYPE-USAGE LATTICE AS PROXY FOR REASONING ON THE NUMBER OF RESPONSIBILITIES OF A CLASS (#CC IS THE NUMBER OF CONNECTED COMPONENTS IN THE TYPE-USAGE LATTICE WITH A THRESHOLD OF 100 TYPE-USAGE SPECIMEN). FOR ALL CLASSES BUT STRING, THE CONNECTED COMPONENTS INDEED REPRESENT CLEAR RESPONSIBILITIES OF THE CLASS.

for Java's String (the first row), we observe in our dataset 69 different type-usages of size 1.

So if one discards those small type-usages, do we still have a large diversity of type-usages? The answer is yes. For $21/30$ classes, there are more than 50% of type-usages whose size is greater or equal to 3 method calls. Those at least 3 method calls are done on the same variable and likely on the same object. Those results show that our empirical data is noisy and that our static analysis indeed capture type-usage fragments. *However, with a conservative assumption that small type-usages are noisy artificial fragments, we still observe a large diversity in API usage.*

VI. Exploiting API Diversity: Reasoning on the Class Semantics using the Type-usage Lattice

We are now confident that, beyond the empirical noise, there exists a large diversity in API usage for some classes. We now want to transform this observational knowledge into actionable knowledge. In this section, we show that the the relation between type-usages can be used as proxy to reason on the class' semantics. As a result, the designers of a class are provided with feedback on the design, and the users are given pieces of documentation that are rarely present in the official documentation.

A. The Lattice of Type-usages

To conduct formal reasoning, we propose to model the type-usages of a given class as a graph. Each type-usage is a node in the graph. The edges should capture the fact that a type-usage is semantically related to another. We model this with a subset relationship. If all the method calls of type-usage x are contained into type-usage y, there is an edge from x to y. By construction, this yields a lattice, since the subset relationship can not be cyclic. Hence we refer to as the lattice of type-usages[5].

Method	*essentiality*	Description
put	0.41	Adds a key-value pair in the map
get	0.29	Gets the value associated with the key
entrySet	0.05	Returns the list of key-value pairs for iterating
...		
getClass	0.0006	Gets the class by introspection (from Object)
wait	0.0001	Tells the current thread to wait (from Object).
notifyAll	0.0001	Wakes up waiting threads (from Object).

TABLE IV

THE MOST AND LEAST ESSENTIAL METHODS OF JAVA'S MAP AS MEASURED BY $essentiality(class, method)$. NOT ONLY THERE IS A LARGE DIVERSITY OF METHOD COMBINATIONS BUT THERE IS ALSO A LARGE DIVERSITY OF METHOD IMPORTANCE.

For example, Figure 8 gives an excerpt of such a lattice for Java's StringBuilder. The visual representation of such those lattices will be discussed in depth in Section VI-D.

If one takes into account all type-usages observed in our dataset, the lattice topology is noisy. For instance, a novice developer may have written an exotic non-meaningful type-usage in one of the applications of our dataset. To remove the noise and have more accurate analyses, the lattice is parametrized with a threshold, which is responsible for filtering out the unimportant type-usages. The threshold is set on $abundance_{ecosystem}(typeusage)$: if a type-usage has been observed at least N times, it is represented, otherwise it is discarded. The rationale is that if a type-usage often appears, it is likely that the corresponding code has been written by many different developers in different context hence is meaningful.

B. Number of Responsibilities

In software engineering, the single responsibility principle (SRP) states that a class should have a single responsibility. It means that all methods of a class should be related to the same single responsibility and work in concert to fulfill it. How does

[5]The formal infimum is a type-usage with no method call, the formal supremum is the set of all methods.

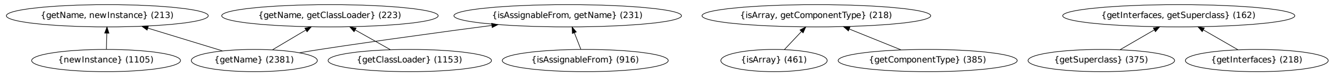

Fig. 6. API Diversity Map of "java.lang.Class". As a piece of documentation, it enables developers to grasp in one glimpse the different responsibilities of the class.

diversity of type-usages relate to this design principle? In this section, we define a metric based on the lattice of type-usages to reason on the responsibilities of a class.

a) Intuition: Our intuition is that the single responsibility principle reflects itself on the type-usage lattice as follows. If a class has one single responsibility, all type-usages are semantically related and the lattice is fully interconnected. If a class has several responsibilities, several groups of semantically-related type-usages emerge, each of them corresponding to a responsibility.

For instance, in the lattice depicted in Figure 6, there are three different separated groups of type-usages that correspond, as we shall see later, to different responsibilities. In classical terms, this can also be seen as a low class cohesion. In other words, we can reason on the class' semantics by analyzing the topology of the lattice of type-usages.

b) Metric:

$$responsibilities(class, threshold) = |cc(typeusages(class))|$$

where cc is the number of the separated connected components in the undirected version of the type-usage lattice; the $threshold$ is the minimum number of type-usage specimen required for a type-usage to be considered in the lattice. The $threshold$ enables us to filter the noisy non-semantic type-usages discussed in V-C.

c) Validation: We compute $responsibilities$ for the 748 most diverse classes of our dataset and a threshold of minimum 100 type-usage specimens. We manually analyze all 8 classes for which there are at least two responsibilities. Those classes correspond to the classes that violate the single responsibility principle. The analysis consists of understanding whether the separated groups of type-usages (each group being a connected component) actually correspond to different responsibilities. This is done based on our own experience as Java developer and on carefully reading the corresponding API documentation.

Table III gives the results of this evaluation. For each of the 8 classes with at least 2 responsibilities, we give the number of connected components in the type-usage lattice and the explanation on their meaning. For instance, the type-usage lattice of Java's interface "Collection" contains 2 connected components: one connected component (cc) is related to iterating over the elements a collection, the other one is about modifying the collection (adding elements). Those two responsibilities make sense according to the API documentation of the class. For "Collection", metric $responsibilities_{ecosystem}$ is validated.

As shown in Table III, the connected components of $7/8$ classes with at least 2 responsibilities make sense and correspond to actual responsibilities. Java's "String" is again an outlier, given a threshold of 100 type-usage specimen by type-usage, the two emerging connected components do not correspond to clear different responsibilities. Interestingly, the API documentation of Java's "Matcher" explicitly mentions at the beginning of the class documentation three responsibilities: our metric identifies with no doubt two of them. For the third one, although it was considered as important as the others at the time of designing and documenting the class, it is much less used in practice. Consequently it does not appear in the filtered type-usage lattice.

We could not check whether all the classes in which there is a single connected component have a single responsibility because of the lack of gold standard. This validation shows that the type-usage lattice enables us to reason on the number of responsibilities of the class. The type-usage diversity is a proxy to the class' semantics.

The diversity of type-usages is actionable, it enables one to reason on the responsibilities of a class.

C. Essentiality of Methods

We leave the level of type-usages and try to reason at the method level directly. We have observed in Section IV-C that not all type-usages are of equal importance. Our goal is to analyze the importance of each method again based on the diversity of type-usages.

We assume that if all methods are of equal importance, then we should find them in similar proportions in type-usages. To reason on this point, we propose the following measure:

$$essentiality(class, meth) = \frac{|\{tus|tus\ contains\ meth\}|}{abundance_{ecosystem}(class)}$$

where *tus* refers to "type-usage specimens" and *meth* is an abbreviation for "method".

The measure $essentiality$ is a ratio between 0 and 1. If $essentiality(c, m)$ is close to 0, it means that few type-usages contain a call to method m and that m is optional. If it is close to 1, it means that most type-usages contain m, hence the method is essential. This measure is the sibling of $frequency$ presented in Section IV-C. While $frequency$ considers type-usages, essentiality focuses on the granularity of methods.

For instance, Table IV gives the essentiality values of methods of Java's Map, which represents a key-value dictionary. The measure captures the most important methods of a Map, the ones that contain the essence of the class: put adds a key-value pair, get retrieves the value associated with a key

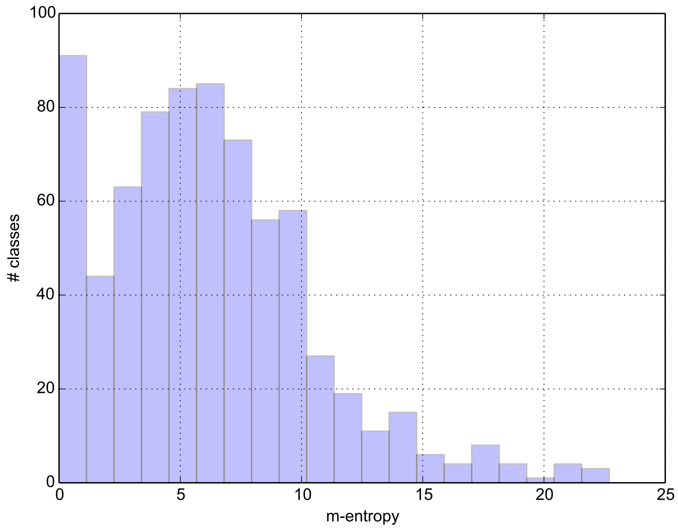

Fig. 7. The distribution of entropy of method essentiality for the most diverse classes.

passed as parameter, $entrySet$ enables one to iterate over all pairs. Similarly, the least important methods come from the root class Object, hence are not specific at all with respect to the semantics of the class.

This measure is actionable. Based on this measure, the designer of a class understands what the real usages of methods are. She can compare the empirical importance against the foreseen usages. For instance, the designer of Java's Matcher envisioned method "lookingAt" as very important and explicitly documented it as such in the API documentation[6]. In practice, less than 1% of all type-usage specimens use this method. Also, the novice user of a class might use this measure for prioritizing the methods she has to learn.

Beyond this practical implication, this measure reflects the diversity of method importance. For all diverse classes of our dataset, the essentiality of methods considerably varies from 0.5 (half of type-usages contain this method) to very small values.

We observe two levels of diversity in API usage, the diversity of method importance (as reflected by $essentiality$) and the diversity of method combinations (as reflected by the $diversity$ and $entropy$).

The measure $diversity$ associates a single number to a class. To analyze the measure $frequency$, we used Shannon's entropy to summarize the distribution of values for each type-usage of a class. Similarly, we propose to measure the entropy of method essentiality, which we call *m-entropy*. If this measure is low, it means that the design of the class relies on a small number of important methods. If it is high, it means that there is a large number of equally important methods.

To some extent, *m-entropy* captures the difficulty of learning a class: if it is high, the user of the class must know many methods, if it is low she can productively work with the class by only knowing a couple of methods. Some consider entropy as a measure of surprise. This is exactly along the same line as difficulty of learning: if most type-usages use the same method, they all are variations around the same goal, which is embodied by the method and there is no surprise. On the contrary, a high *m-entropy* means that the developer would regularly be surprised by a type-usage that contains a new method and no already known method. For instance, the *m-entropy* of Java's String (the most diverse class of our dataset) is 1.1, which is low compared to other diverse classes. This fits to the experience of Java developers that String is not a class that is complex to understand and use.

Figure 7 shows the distribution of entropy of method essentiality for all the 748 most diverse classes of our dataset. We observe interesting phenomena on this figure. First, there are two modes. There is a pack of classes with an m-entropy ≤ 2. Despite diverse in their method combinations, those classes are easy to learn because they are built one or two central methods. Then, there is a maximum density of classes for classes around m-entropy $= 7$. Let us take again a concept from ecology to explain this phenomenon. This tend to show that there is a sweet spot in terms of design, a kind of ecological niche where many classes converge. An open intriguing question is: what does this value of 7 mean? Future work might answer this question by proposing and comparing different generative models of API usage.

Finally, the classes with the maximum *m-entropy* culminate at m-entropy ≥ 20. First , many of those classes are generated, and we find in particular many generated parsers. Those classes are not "natural", and this is reflected in the high artificial *m-entropy*. But beyond those outliers, we observe that this average maximum entropy is higher than the entropy of type-usages presented in Section IV-D where the maximum values were around 10.

There are two drivers in entropy computation in discrete spaces: the number of considered elements and the uniformity of the distribution: the entropy is proportional to the number of elements (the number of methods in this case), and to the uniformity (a uniform distribution yields maximum entropy as discussed in IV-D). For all classes under consideration, there are much more type-usages than methods (see Table II). Consequently, since *m-entropy* has higher values than *u-entropy*, it means that the distribution of essentiality is much more uniform and that methods less dominate the distribution than for type-usages. This indicates that method combinations are not randomly chosen based on the importance of methods but that there is some kind of structure behind the combinations: if methods a and b both have an essentiality of 0.3 (they appear in 30% of all type-usage specimens), it does not mean that one observe X types usages with only a and X with only b. Methods a and b may frequently occur together and peak as a single dominating type-usage. In this case, this is reflected by *u-entropy* higher than *m-entropy*.

Intuitively, software design is analyzed with discrete concepts. For instance, the 6 metrics of Chidamber and Kemerer

[6] http://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html

for analyzing object-oriented design [8] are all discrete[7]. The reason might be that the basic elements of software are either binary or enumerated. However, the analysis we have presented in this section lets us think it makes sense to reason on the design of real classes with a continuous conceptual framework. In real classes, there are dozens of methods, adding or removing methods, even many does not make any significant difference on the design quality as long the design of responsibilities remains consistent. In this case, the number of methods, which is discrete, is less meaningful than the *m-entropy*. What matters is that the class is still built around one or two clear flagship methods and a very continuous concept using probabilities (entropy) seems to capture this design property.

D. Visual Representation of Usage Diversity

We propose to use the lattice of type-usages as a piece of documentation. An "API diversity map" is a graphical representation of the lattice, laid out so that the largest type-usages (in number of method calls) are at the top and the smallest at the bottom. The filtering threshold on the abundance enables one to tune the size of the API diversity map.

Figure 8 gives the diversity map of Java's StringBuilder. The values for each type-usage correspond to $abundance_{ecosystem}(typeusage)$. StringBuffer is a class used for manipulating strings in an efficient manner.

The threshold on a minimum abundance of 150 specimen per type-usage results in 8 nodes which makes it very readable. The unfiltered noisy lattice would contain $diversity(StringBuilder) = 643$ different nodes. This map is very layered, due to the semantics of edges ("subset of"). One sees that there is a "master" type-usage in which all common methods of StringBuffer are used ("init" refers to a constructor call). One also sees that some type-usages are more popular than others. For instance, {init, append, toString} appears 2434 in our dataset. For developers who know StringBuilder, this reflects well its different usages. For instance, on one end of the usage spectrum, one often only calls "append" on a StringBuilder passed as parameter. On the other end of the usage spectrum, one uses all main methods of StringBuilder in a same method.

Now consider the diversity map of Java's "Class" represented in Figure 6, the class handling the reflection of any object (the meta-object is obtained by calling "getClass"). Compared to the diversity map of StringBuilder, we observe that: first the map is divided in three separated trees (the different responsibilities already discussed); second, the top layer of the map is composed of 5 different type-usages. Both phenomena are due to the fact that Java's "Class" has different responsibilities: creating objects ("newInstance"), proxying the current thread's class loader ("getClassLoader"), testing instance-of relationships ("isAssignableFrom"), handling Java array special semantics ("isArray"), and subtyping introspection ("getInterfaces, getSuperClass").

[7]Only WMC may not be discrete depending on how one compute the complexity)

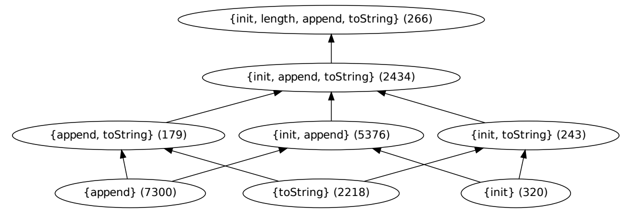

Fig. 8. API Diversity Map of "java.lang.StringBuilder". The numbers in bracket is $abundance_{ecosystem}(typeusage)$.

API diversity maps make diversity actionable. Based on the maps we analyzed, they convey in one glimpse the usage spectrum of class. This may be valuable for both the designers and the new users of a class.

VII. Discussion

We have observed a large-scale diversity in the usage of object-oriented classes. To what extent, does this phenomenon impact our software engineering knowledge? In particular, what are the reasons behind this diversity? what are the implications of this diversity? In this section, we speculate about those two points, reasons and implications in order to identify new fruitful research directions.

A. Speculative Reasons of API Diversity

1) Diversity and Cognition: When programming with object oriented APIs, the bulk of the cognitive load consists of remembering identifiers related to tasks (whether package, class or methods). With this respect, remembering one single class name is easier than remembering three of them. If Java's String would have been split in several classes, each one handling one fine-grain responsibility (one subset of type-usages), this would have increased the cognitive load of developers. This argument applies to all classes and is related to research on API usability, in which we have not found studies about diversity. This argument would mean that, in terms of object-oriented API design, there is a trade-off between responsibility decomposition and usability. We think that future research on this point would be of great interest.

2) Diversity and Plasticity: Second, let us define "class plasticity" as the ability of a class to be used in many different ways. Many factors influence the "class plasticity". First, we have seen that the number of public methods increases the number of possible method call combinations, hence is correlated with the plasticity (although slightly as witnessed by the Spearman coefficient). Second, all kinds of checks have an impact on the plasticity as well. For instance, overly restrictive pre-condition and post-condition checks hinder plasticity. We tend to think that a high usage diversity reflects a high class plasticity.

3) Diversity and Reusability: High usage diversity may correlate with reusability. It can reflect the fact that client code was able to use the class in ways that were unanticipated by the class designer. For instance, if one high level method is defined on three sub-routines, providing the subroutines as public would probably provoke unanticipated reuse of those routines, which would consequently increase the class API usage diversity. Having maps of API diversity as proposed in VI-D may guide reuse. With those maps, developers are aware of whether certain type-usages are popular or not and can make informed decisions on how to use a class.

4) Diversity and Immutability: It is to be noted that one can add as many public methods to an immutable object without breaking anything: there are neither state-changing risks nor usage protocol issues. In other terms, an immutable class easily gives birth to a high API usage diversity. Java's String being immutable, this argument probably contributes to the massive usage diversity we have observed.

5) Diversity and Success: Innovators try to write "successful code". In a commercial perspective, to make a lot of money; in an open-source perspective , to gather a lot of users. For an object-oriented library, "successful" means having many client pieces code. For a class, "successful" means having many client type-usages across many different software projects. Certain classes of the Java Development Kit are successful, as are classes of external libraries (e.g. the Apache Commons libraries).

How to write successful classes? There is no clear recipe and there are probably many factors influencing the success: technical, social and commercial. However, it is generally accepted that a badly designed class has little chances to survive and become popular.

We have observed many classes that are successful (widely used across a large ecosystem), and that have a large number of public methods as well as a large diversity of possible different usages. Even if those characteristics are sometimes considered as bad design (as a violation of the single responsibility principle aforementioned), they did not prevent those classes to become successful. This holds for JDK classes as well as for non JDK classes (e.g. W3C's Node). To sum up, according to our results, *a high API usage diversity does not prevent success*.

We are also tempted to go further: *if a class supports a high API usage diversity, it may favor its success*. The following section presents arguments in favor of diversity in API design.

B. Speculative Implications of API Diversity

We have just discussed development practices that could explain the emergence of high degrees of usage diversity. In this section we discuss the impact of such diversity on several aspects of software quality.

1) Diversity and Testability: Object-orientation has been a major concern in the software testing community: does it favor or hinder error finding? In particular, increased encapsulation, modularity and coupling issues brought by the object-oriented paradigm led to a large amount of work that discuss the impact on testability [5], [1], [20]. Today, there is no doubt about the utility of object-orientation, and testers have found effective ways to reveal and fix errors in object-oriented code. However, the observations that we make in this paper seem to raise new questions about testability and maintainability of object-oriented libraries. How to ensure that all possible type-usages are correct? Should there be one test per observed API usage (i.e. 2 460 test cases for Java's String), or even one test per acceptable method call combinations? This highlights a particularly intriguing relation between diversity and oracles, which we would put as diversity and correctness. Does API usage diversity reflect a fuzzier notion of correctness? Does API usage diversity means that we can only have "partial" oracles? This is an open question calling for future research on software testing.

2) Diversity and Bug Detection: The type-usage abstraction has been introduced for sake of static bug detection [18], [19]. In this previous research, our mantra was to find a definition of "anomaly" among type-usages, a definition that yields a low number of false positive. An intuitive threshold on the abundance, even drastic, does not work. However, we achieved a false positive ratio to the price of adding strong criteria in the definition of "type-usage anomaly": first, with respect to the context of the type-usage (the enclosing method), second, with respect to a type-usage distance expressed in terms of methods calls. The new results presented in this paper illuminate our previous work: the diversity of type-usages makes it impossible to easily define an "anomaly". When an observed world is too diverse, there is no such thing as "anomaly" or "out of the norm". In general, we tend to think that the more diversity in code (resp. at runtime), the less possible it is to define high confidence static (resp. dynamic) bug detection rules.

3) Diversity and Repair: However, beyond bug detection, for automated bug repair, diversity may also as be a major opportunity. The existence of a large number of similar, yet diverse type usages provides a wonderful 'reservoir' of alternative code to fix bugs. This goes in the direction of recent results by Carzaniga and colleagues [7] showing that the API usage diversity and plasticity can be used to fix certain bugs at runtime. In such cases, the diversity gives a kind of mutational robustness [23].

4) Diversity and Diversification: In this work we make original observations about the presence of large scale diversity in software. This diversity is present and has emerged spontaneously through the development of a large number of Java classes. One question that emerges with the observation of this spontaneous emergence of diversity is: should we support or encourage the diversity in object-oriented software? Beyond the impact of diversity on success discussed in VII-A5, what about inventing techniques that automatically diversify a class API, using novel code synthesis mechanisms?

For example, let us imagine a developer who wants to use a class X. The developer calls a number of methods of this class' API, based on previous experiences with this API and a rather intuitive comprehension of what this class should do. There is a chance that the developer calls a method that is not part of the API, but that relates to the services offered by this API. If this case happens, there may be a possibility that the yet unknown method can be implemented as a combination of existing methods. One way to automatically diversify a class API would be to automatically synthesize this new method, using the code provided by the developer as the specification (if the code executes correctly, the generated method is correct). This kind of code synthesis would, by definition, increase the diversity of type usages over the API, and its principles would be similar to the theories underlying mediator synthesis for middleware interoperability [4], [6].

C. Recapitulation

We think that our observations on object-oriented API usage diversity have questioned different parts of the software engineering knowledge in particular with respect to the principles of good API design. We also think that it opens new research questions in terms of API usability and software testing.

VIII. Related Work

Gabel and Su [10] have studied the uniqueness and redundancy of source at the level of tokens. Our study explores a different facet at a difference granularity: the diversity at the level of object-oriented type usages.

Baxter et al. [3] have studied the "shape" of Java software. They discuss the empirical distribution of many software metrics, in particular size based metrics. However, they don't discuss at all diversity metrics as we do in this paper.

At the level of object-oriented APIs, an early paper by Michail [17] discusses object-oriented usage patterns that were observed in a large-scale study. He did not mention "diversity" although it was somehow implicit in the large reported number of patterns mined (51308 only for KDE classes). On the contrary, we focus on measuring, analyzing and understanding this diversity.

Ma and colleagues [13] only focus on Java classes and prevalence metrics. Laemmel et al. [12] talk about API footprint and coverage (the number of API classes and methods used within client projects). They do not mention the usage diversity.

To our knowledge, Veldhuizen [25] is the only one who has looked at entropy in software in a similar meaning as we have. However, his point on entropy and reuse is more theoretical than empirical, and the presented results are at the level of low-level C library. To our knowledge, we are the first to report on the existence, with precise numbers, of large scale diversity at the API usage level.

Recently, Posnett et al. [21] explored a facet of diversity in software development. In their paper, they define the notions of "artifact diversity" and "authorship diversity" and extensively discuss the pros and cons of high diversity. For instance; for a module, it is beneficial to have a high diversity of contributors. Posnett et al. and we both specifically aim at measuring and understanding diversity in software. But we focus on different facets: "artifact diversity" and "authorship diversity" are orthogonal to "API usage diversity".

IX. Conclusion

We have mined 9 022 262 type-usages in 3 418 Jar files totaling 382 774 Java classes. In this data, we wanted to specifically measure the *diversity*, in the sense of ecological biodiversity. To our surprise, we observed a large-scale usage diversity of API usage: 748 classes are used in more than 100 different ways. To our knowledge, this phenomenon has never been reported before.

Then, we have put this diversity to work. We have shown how to use the diversity of API usages as proxy to reason on a class' semantics, for instance to reason on the number of responsibilities. Finally, we have discussed those empirical results in the general context of software engineering: what are the reasons behind this diversity? what are the implications of this diversity?

As future work, it would be interesting to define measures of "diversity" at other levels of abstraction (e.g. tokens or control flow structures) to analyze the scale effect of this software metric [22]. Diversity may also vary depending on the application domains, and programming languages. To conclude, the diversity advocated by Stephanie Forrest [9] may have already emerged at many layers of the software stack and this work provides new empirical insights about this phenomenon.

Acknowledgments

This work is partially supported by the EU FP7-ICT-2011-9 No. 600654 DIVERSIFY project and the INRIA Internships program. We thank Benoit Gauzens for detailed feedback as well as Yann-Gaël Guéhéneuc, Vivek Nallur and all members of the DIVERSIFY project for insightful discussions.

References

[1] B. Baudry, Y. Le Traon, and G. Sunyé. Testability analysis of a UML class diagram. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 54–63. IEEE, 2002.

[2] B. Baudry and M. Monperrus. Towards Ecology-Inspired Software Engineering. *arXiv preprint arXiv:1205.1102*, 2012.

[3] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. Understanding the shape of Java software. In *Proceedings of Object-oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2006.

[4] G. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. *Middleware 2011*, pages 410–430, 2011.

[5] M. Bruntink and A. Van Deursen. Predicting Class Testability using Object-oriented Metrics. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 136–145. IEEE, 2004.

[6] C. Canal, P. Poizat, and G. Salaun. Model-based Adaptation of Behavioral Mismatching Components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.

[7] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic Recovery from Runtime Failures. In *Proceedings of ICSE'13*, 2013.

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

[9] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[10] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

[11] I. J. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3-4):237–264, 1953.

[12] R. Lämmel, E. Pek, and J. Starek. Large-scale, AST-based API-usage Analysis of Open-source Java Projects. In *SAC'11 - ACM 2011 Symposium on Applied Computing, Technical Track on "Programming Languages"*, 2011.

[13] H. Ma, R. Amor, and E. Tempero. Usage Patterns of the Java Standard API. In *Proceedings of APSEC*, pages 342–352. IEEE, 2006.

[14] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[15] D. Mendez, B. Baudry, and M. Monperrus. Companion Web Page for "Empirical Evidence of Large-Scale Diversity in API Usage of Object-Oriented Software". http://www.monperrus.net/martin/companion-diversity-api-usages, 2013.

[16] D. Mendez, B. Baudry, and M. Monperrus. Empirical evidence of large-scale diversity in api usage of object-oriented software. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2013.

[17] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the International Conference on Software Engineering*, pages 167–176, 2000.

[18] M. Monperrus, M. Bruch, and M. Mezini. Detecting Missing Method Calls in Object-Oriented Software. In *Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer, 2010.

[19] M. Monperrus and M. Mezini. Detecting Missing Method Calls as Violations of the Majority Rule. *ACM Transactions on Software Engineering and Methodology*, 22(1), 2012.

[20] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 84–93. IEEE, 2001.

[21] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. Dual Ecological Measures of Focus in Software Development. In *Proceedings of ICSE*, 2013.

[22] D. Posnett, V. Filkov, and P. T. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of ASE*, pages 362–371, 2011.

[23] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software Mutational Robustness. *arXiv preprint arXiv:1204.4224*, 2012.

[24] R. Vallée-Rai, L. Hendren, V. Sundaresan, E. G. Patrick Lam, and P. Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[25] T. L. Veldhuizen. Software Libraries and their Reuse: Entropy, Kolmogorov Complexity, and Zipf's Law. *arXiv preprint cs/0508023*, 2005.