How Annotations are Used in Java: An Empirical Study

Henrique Rocha*, Marco Tulio Valente*

*Department of Computer Science
Federal University of Minas Gerais (UFMG)
Email: henrique.rocha@gmail.com, mtov@dcc.ufmg.br

***Abstract*—Since 2004, Java provides support to general purpose annotations (also known as metadata) that allows developers to define their your own annotation types. However, seven years after their inception in the Java language, we still do not have empirical evidence on how software developers are effectively using annotations in their systems. Therefore, this paper presents an empirical study on how annotations are used on a corpus of 106 open-source Java systems. On total, we have evaluated more than 160,000 annotations that have been applied to the source code of such systems. Our main findings can be summarized as follows: (a) the so-called annotation-hell phenomena affects many of the evaluated systems; (b) developers are using both pre-defined annotations and annotations defined by external frameworks, mostly annotations dedicated to persistence and testing; (c) most of the evaluated annotations have been employed to annotate methods (more than 90%); (d) although Java does provide not support to annotations for anonymous classes, several programs from our corpus have applied annotations to such classes.**

***Index Terms*—Annotations, Empirical Studies, Qualitas Corpus.**

I. Introduction

Annotations have been one of the features introduced in J2SE 5.0 released in 2004 [3], [1]. Annotations are a metadata feature that provides the ability to associate additional information to Java elements (like classes and methods). Annotations have no direct effect on the code they annotate, and they do not change the way in which the source code is compiled. There are at least the following possible uses for annotations [3], [1], [6]:

- Compiler information (using compiler annotations) to suppress warnings and to detect errors. For example, developers can use the `@SuppressWarning("unchecked")` annotation to suppress the unchecked warning that can be raised when using an old code written before J2SE 5.0.
- Generation of additional files such as source code, XML, help files, and so on. For example, developers can annotate some system's functions with `@Help("...")` and later an annotation processing tool can read this metadata and generate a help file.
- Source code documentation by javadoc and similar tools (using the `java.lang.annotation.Documented` annotation).
- Since annotations can be processed at runtime, they can be used for logging, testing and other utilities.
- Some APIs (like Java Persistence API) relies on annotations as a declarative way to provide their services. Instead of calling methods or declaring objects, an annotation is placed in the program elements that the developers want to be annotated. For example, to define a class as a `JPA` entity, we just need to put the `@Entity` annotation on the class.

Although annotations have several useful applications, their use could massively contribute to code pollution. This phenomenon is usually called *annotation hell*, i.e. the source code becomes cluttered with so many annotations leading to poor code readability and understandability.

The ultimate goal of our research is to empirically investigate the use of annotations in Java programs. Annotations have been a part of the Java language for almost seven years, and now it is reasonable to assume that they are being used in most Java applications. We are particular interested in providing answers to the following questions: (a) Which are the most used annotations? (b) Which program elements (i.e. methods, classes) are annotated more frequently? (c) Are there many systems suffering from the *annotation hell* phenomenon?

In order to answer such questions, we present the results of an in-depth empirical study including the inspection of annotations in 106 open-source Java systems that are a part of the Qualitas Corpus. The Qualitas Corpus is a collection of open-source Java sytems designed to help code analysis in empirical studies [9], [8]. Basically, we developed two approaches to extract the annotation data from the Qualitas Corpus. The results from both approaches are shown and compared, providing a better analysis to the use of annotations and even showing some tendencies that programmers are following.

The remainder of this paper is organized as follows. In Section II, we provide an introduction to the use of annotations in Java, explaining their most recomended uses and underlying syntax. Section III describes the methodology used for the empirical study, including the two approaches we have followed to retrieve annotations in Java systems. Section IV analyzes and discuss the obtained results. In Section V we present this study's threats to validity. Section VI discusses related work and Section VII concludes the paper, summarizing its main

contributions and outlining future work.

II. Annotations

Annotations are a metadata feature that does not directly affect the code they annotate. As mentioned in the Introduction, they can be used for several purposes like runtime testing, generating additional files, providing information to compilers, and providing a simple and declarative form to use API features [3], [1], [6].

In terms of implementation, annotations can be considered a special kind of interface [3], [1]. To distinguish an annotation declaration from an interface, annotations are declared using the `@interface` keyword. Listing 1 shows a code fragment that declares an annotation called `Author`. Annotations can make use of access modifiers just like interfaces (`public`). In the example, the `Author` annotation has the *element* `name` which has a `String` type and by default has the `"[unassigned]"` value. Annotations *elements* are a special kind of method declaration.

```
1 public @interface Author {
2     String name() default "[unassigned]";
3 }
```

Listing 1. Code fragment declaring an Annotation

Listing 2 shows how the `Author` annotation can be used to annotate a class. This code also shows some `JPA` annotations. For example, the `@Entity` denotes that the `Customer` class will be made persistent by `JPA`. The `@Id` annotation is used to define the entity's primary key.

```
1 @Author{name="Henrique"} @Entity
2 class Customer implements Serializable{
3
4     @Id
5     public Integer getCustomerId(){
6       ...
7     }
8 ...
```

Listing 2. Annotations Example

Annotations can be divided based on their retention policy into three types: Source, Class and Runtime annotations [3]. Source annotations are processed during compilation (usually by a compiler plug-in), but they have no effect on the generated code. Class annotations are also processed during compilation and they are stored in the generated class files. Runtime annotations are stored in class files and can be recovered at runtime through the Java Reflection API.

There are some predefined annotations in Java, including the following compiler annotations: `@Deprecated`, `@Override`, and `@SuppressWarning`. The `@Deprecated` annotation indicates that the marked component is deprecated and should no longer be used. The `@Override` is an annotation for methods only, and it indicates that the method is supposed to override a superclass method. Finally, the `@SuppressWarning` annotation disables a specific warning the compiler would otherwise generate.

The additional data provided by annotations can be processed by the compiler, other tools or, in the case of runtime annotations, they can be examined at runtime using reflection. The most common tool to work with annotations is the annotation processing tool (`apt`) provided in the Java development toolkit. The `apt` retrieves annotations in source files, using custom factories and visitors so that programmers may process annotations according to their needs.

III. Methodology

In this section we describe the target systems used in the study (Section III.A) and the methodoly we followed in the research (Section III.B).

A. Target Systems

We defined the scope of our study to be the systems contained in the Qualitas Corpus [9]. The Qualitas Corpus is a *curated* collection of open-source Java systems whose ultimate goal is to reduce the effort in finding, obtaining, and organizing code sets to be used in empirical studies [9]. We used the Qualitas Corpus version `20101126r`, that contains only the most recent versions of the 106 systems.

B. Study Design

To retrieve the annotations used by the systems in the Qualitas Corpus, we implemented a factory class for the `apt` tool, that is a part of the Java SE Development Kit (JDK) version 1.6.0 update 24. The `apt` parses the source files and provides to the factory class each annotation found, as well as details about the annotation. Basically, our factory class saves all the relevant information about the annotations in a relational database for latter analysis.

Since `apt` is the official Java tool to process annotations, our original plan was to use it to examine all annotations used by the systems included in the Qualitas Corpus. However, the `apt` tool has stopped with runtime errors while trying to process 15 systems – Table I shows the name of these systems. The errors occurred on the tool itself, and even bug report information has been raised to contact the manufacturer.

TABLE I
Qualitas Corpus systems the APT has not been able to Process

System	System	System
Castor	javacc	nakedobjects
Cayenne	jboss	netbeans
gt2	jrefactory	springframework
Hibernate	maven	struts
jFin_DateMath	myfaces	tapestry

Due to the `apt` errors, we created a textual search program to find annotations in source files. Originally, this textual search approach has been used on the systems that the `apt` tool has not been able to process. Latter we expanded the textual search to analyze all the systems to better compare the differences between both approaches.

TABLE II
ANNOTATIONS FOUND ON THE QUALITAS CORPUS

System	KLOC	APT		Textual Search	
		Total	Density	Total	Density
ant	107.770	4	0.04	4	0.04
antlr	25.243	984	38.98	995	39.42
aoi	111.725	21	0.19	21	0.19
argouml	194.859	2038	10.46	2047	10.51
aspectj	412.394	889	2.16	893	2.17
azureus	453.433	39	0.09	40	0.09
castor	115.543	–	–	929	8.04
cayenne	127.529	–	–	3473	27.23
checkstyle	23.316	1413	60.60	1529	65.58
cobertura	51.860	12	0.23	12	0.23
compiere	400.257	3621	9.05	3752	9.37
derby	592.817	22	0.04	22	0.04
drjava	62.380	123	1.97	177	2.84
eclipse_SDK	2282.511	798	0.35	916	0.40
findbugs	109.096	2122	19.45	2387	21.88
fitlib	27.539	625	22.70	658	23.89
freecol	81.671	205	2.51	278	3.40
freecs	23.012	5	0.22	6	0.26
gt2	446.863	–	–	7001	15.67
heritrix	61.681	150	2.43	177	2.87
hibernate	163.858	–	–	6755	41.22
hsqldb	123.268	22	0.18	22	0.18
htmlunit	40.004	5881	147.01	5959	148.96
informa	9.722	3	0.31	3	0.31
ireport	221.490	5454	24.62	5549	25.05
itext	76.369	371	4.86	371	4.86
jFinDateMath	4.807	–	–	64	13.31
jasperreports	170.064	99	0.58	106	0.62
javacc	13.772	–	–	8	0.58
jboss	281.643	–	–	2582	9.17
jchempaint	90.831	8923	98.24	8923	98.24
jedit	107.469	486	4.52	523	4.87
jena	70.948	1148	16.18	1474	20.78
jgrapht	11.931	53	4.44	53	4.44
jgroups	96.325	1436	14.91	1436	14.91
jhotdraw	75.958	3143	41.38	3570	47.00
jmeter	81.010	1164	14.37	1234	15.23
jre	914.867	1182	1.29	1197	1.31
jrefactory	113.427	–	–	44	0.39
jruby	160.360	5139	32.05	5217	32.53
jspwiki	43.326	158	3.65	160	3.69
jung	37.989	422	11.11	432	11.37
junit	6.164	171	27.74	213	34.56
lucene	113.223	106	0.94	112	0.99
marauroa	13.823	424	30.67	434	31.40
maven	54.336	–	–	880	16.20
megamek	258.957	1649	6.37	1839	7.10
myfaces_core	119.529	–	–	2844	23.79
nakedobjects	110.378	–	–	4578	41.48
netbeans	1890.536	–	–	57820	30.58
picocontainer	9.259	196	21.17	206	22.25
pmd	60.875	769	12.63	786	12.91
poi	143.507	286	1.99	294	2.05
proguard	55.567	45	0.81	45	0.81
quartz	26.819	32	1.19	33	1.23
roller	50.980	96	1.88	97	1.90
rssowl	73.230	1639	22.38	2434	33.24
spring	160.302	–	–	7883	49.18
squirrelsql	6.944	4	0.58	15	2.16
struts	74.670	–	–	1848	24.75
sunflow	21.648	17	0.79	25	1.15
tapestry	53.367	–	–	4263	79.88
tomcat	166.478	2717	16.32	2897	17.40
trove	2.196	3	1.37	4	1.82
weka	224.356	21	0.09	21	0.09
Total		56330		160570	

IV. RESULTS

In this section we present the results from our study. Table II shows the systems, their Kilo non-comment Lines of Code (KLOC), the total annotations found and the annotation density as retrieved by the `apt` tool and by the textual search. The annotation density is calculated by dividing the number of annotations by the KLOC. Therefore, this density represents an average of how many annotations are present in every one thousand lines of code. As mentioned, the `apt` tool crashed with an internal bug on 15 systems, these systems are shown on the table with a "–".

Two more aboservations should be made about the data in Table II, first the table shows that the size of the evaluated systems has no impact on the runtime errors raised by the `apt` tool. For example, the `castor` system has 115.543 KLOCs and caused a crash in the `apt`, while the `Eclipse_SDK` system has 2,282.490 KLOCs and has been sucessfully processed.

Second, Table II also shows a slight difference between the number of annotations found by the `apt` and by the textual search. Investigating further, we discovered that Java does not support annotations in the scope of anonymous classes. Because of that, the `apt` has completely ignored those annotations. On the other hand, the implemented textual search has count such annotations.

To better inspect the different measurements regarding annotations in anonymous classes, we will use the `jhotdraw` system as an example. As can be observed on Table III, the measurements have a difference of 427 annotations in their count. Table III also shows the particular annotations present in the `jhotdraw` system. As we can observe, the main difference is in the `@java.lang.Override` annotation, followed by the `@java.lang.SupressWarning` annotation. In fact, this seems to be case not only for the `jhotdraw`, but for every system in the Qualitas Corpus with a different annotation count between the `apt` and the textual search.

TABLE III
JHOTDRAW ANNOTATIONS

Annotation	APT	Textual Search
@java.lang.Override	2909	3323
@java.lang.SuppressWarnings	89	102
@org.jhotdraw.annotations.Nullable	95	95
@org.jhotdraw.annotations.NotNull	42	42
@java.lang.Deprecated	4	4
@java.lang.annotation.Documented	2	2
@java.lang.annotation.Target	2	2
Total	**3143**	**3570**

From all the 106 analyzed systems, 41 did not have a single annotation – Table IV shows these systems and the year of their release date. Analysing the years on this table, we can

conclude that seven systems were release before 2004 (i.e. before the introduction of annotations in Java) and another seven systems have been release in 2004.

TABLE IV
QUALITAS CORPUS SYSTEMS WITHOUT ANNOTATIONS

System	Year	System	Year	System	Year
axion	2003	jasml	2006	nekohtml	2010
c_jdbc	2005	jext	2004	openjms	2007
colt	2004	jfreechart	2009	oscache	2007
columba	2005	jgraph	2009	pooka	2008
displaytag	2008	jgraphpad	2006	quickserver	2006
drawswf	2004	jmoney	2003	quilt	2003
emma	2005	joggplayer	2002	sablecc	2005
exoportal	2006	jparse	2004	sandmark	2004
fitjava	2004	jpf	2007	velocity	2010
galleon	2006	jrat	2003	webmail	2002
ganttprj	2009	jsXe	2006	xalan	2007
ivatagrp	2005	jtopen	2010	xerces	2010
jag	2006	log4j	2010	xmojo	2003
james	2004	mvnforum	2010		

Finally, Table V shows the top ten Qualitas Corpus systems with annotations using both measurements. Based on this table it is possible to conclude that some of the systems the `apt` was not able to process have a large number of annotations, for example, `netbeans`, `spring framework` and `hibernate`.

TABLE V
TOP TEN QUALITAS CORPUS SYSTEMS WITH THE HIGEST NUMBER OF ANNOTATIONS

APT			Textual Search		
System	Total	Density	System	Total	Density
jchempaint	8923	98.24	netbeans	57820	30.58
htmlunit	5881	147.01	jchempaint	8923	98.24
ireport	5454	24.62	spring	7883	49.18
jruby	5139	32.05	gt2	7001	15.67
compiere	3621	9.05	hibernate	6755	41.22
jhotdraw	3143	41.38	htmlunit	5959	148.96
tomcat	2717	16.32	ireport	5549	25.05
findbugs	2122	19.45	jruby	5217	32.53
argouml	2038	10.46	nakedobjects	4578	41.48
megamek	1649	6.37	tapestry	4263	79.88

In the remainder of this subsection we first discuss the results using the `apt` tool (Section IV.A). Second, we show the results provided by the textual search approach (Section IV.B). And finally, we discuss both results in order to clarify our main findings (Section IV.C).

A. APT

Table VI shows the top ten analyzed systems with the highest annotation density. For example, the `htmlunit` system – the system that presented the highest annotation density – has a 147.01 density, which means that on average on each one thousand lines of code there will be 147 annotations.

Table VII show the total number of annotations from all systems processed by the `apt` classified by program element. In this classification, a type denotes packages, classes, enums or interfaces. As we can see, more than 92% of the annotations are used to annotate methods.

TABLE VI
TOP TEN QUALITAS CORPUS SYSTEMS WITH HIGHEST ANNOTATION DENSITY AS PROCESSED BY THE APT

System	KLOC	Annotations	
		Total	Density
htmlunit	40004	5881	147.01
jchempaint	90831	8923	98.24
checkstyle	23316	1413	60.60
jhotdraw	75958	3143	41.38
antlr	25243	984	38.98
jruby	160360	5139	32.05
marauroa	13823	424	30.67
junit	6164	171	27.74
ireport	221490	5454	24.62
fitlib	27539	625	22.70

TABLE VII
ANNOTATIONS BY PROGRAM ELEMENT

Element	Annotations	Percentage
Type	2459	4.36
Constructor	612	1.09
Field	865	1.54
Method	51931	92.19
Parameter	463	0.82
Total	**56330**	**100**

Finally, Table VIII presents the most used annotations in the processed systems. As expected, the three compiler annotations are used very frequently in such systems.

TABLE VIII
MOST USED ANNOTATIONS IN THE QUALITAS CORPUS SYSTEMS PROCESSED BY APT

Annotation	Number	Percentage
@java.lang.Override	28723	50.99
@Test	11327	20.10
@org.jruby.anno.JRubyMethod	2762	4.90
@org.openscience.cdk.annotations.TestMethod	2735	4.85
@java.lang.SuppressWarnings	2699	4.79
@java.lang.Deprecated	1783	3.16
@com.grlsoft.htmlunit.BrowserRunner.Alerts	1523	2.70
@org.openscience.cdk.annotations.TestClass	443	0.78
@RunWith	359	0.63
@org.jgroups.annotations.Property	290	0.51
@BeforeClass	279	0.49

B. Textual Search

Table IX shows the ten analyzed systems with the highest annotation density, as processed by our text-based search approach. By comparing Tables VI and IX, it is possible to conclude that some systems the `apt` has not been able to process have a high density value and they appeared on the textual search (`tapestry`, `spring framework`, `nakeobjects`, `hibernate`). The system that presented the highest and second highest densities were respectivelly the `htmlunit` and `jchempaint`.

Due to limitations inherent to the textual search approach, we could not determinate to which program element the annotations belong to. Finally, Table X presents the most used annotations in all systems. We can observe that, as in the `apt` tool based results, the compiler annotations are widely used.

TABLE IX
TOP TEN QUALITAS CORPUS SYSTEMS WITH THE HIGHEST ANNOTATION DENSITY AS PROCESSED BY THE TEXTUAL SEARCH

System	KLOC	Annotations	
		Total	Density
htmlunit	40.004	5959	148.96
jchempaint	90.831	8923	98.24
tapestry	53.367	4263	79.88
checkstyle	23.316	1529	65.58
springframework	160.302	7883	49.18
jhotdraw	75.958	3570	47.00
nakedobjects	110.378	4578	41.48
hibernate	163.858	6755	41.22
antlr	25.243	995	39.42
junit	6.164	213	34.56

TABLE X
TEXTUAL SEARCH ANNOTATIONS MOST USED IN THE CORPUS' SYSTEMS

Annotation	Number	Percentage
@Override	92275	57.46
@Test	20875	13.00
@SuppressWarnings	7714	4.80
@Deprecated	3377	2.10
@Column	2836	1.76
@JRubyMethod	2755	1.71
@TestMethod	2735	1.70
@Alerts	1509	0.93
@Entity	1121	0.69
@Id	951	0.59

C. Analysis

Based on the results presented on this paper we can present some interesting findings about the use of annotations.

First, by inspecting Tables VI and IX, we can observe that the annotation density values for these systems (specially those on Table IX) are a strong indication they might be suffering from *annotation hell*. For example, when we analyze the `htmlunit` system, it scored the highest density in both approaches, having almost 150 annotations per KLOC (or approximatly one annotation at 7 lines of code).

A second interesting finding is about the annotations used on the systems. We initially expected that all three compiler annotations to be amongst the most used ones. But the second most used annotation was the `@Test` annotation. The `@Test` annotation is used to test methods dynamically through reflection. Apparently, this has become a popular standard used by several programmers.

Moreover, we can see on Table X that the annotations `@Column`, `@Entity` and `@Id` are used very frequently. These annotations are used by the Java Persistence API to map Java classes to relational database tables.

Another annotation used frequently, as presented on Tables VIII and X, is the `@org.jruby.anno.JRubyMethod`. This annotation is used by the `jruby` system to specify methods signatures using Java code and to use them as method calls in Ruby code. To illustrate the use of `@JRubyMethod`, Listing 3 shows an example of a `readchar` method marked with this annotation. As mentioned, the goal is to mark this method callable from code within Ruby.

```
1 @JRubyMethod(name = "readchar")
2 public static IRubyObject readchar(
      ThreadContext context, IRubyObject recv)
      {
3 ...
```

Listing 3. JRubyMethod Annotation Example

A third interesting finding comes from the observation of Table VII. As we can see, most annotations were used to annotate methods (over 90%), which is a huge difference to the second most used annotation, the types annotations with almost 5%. Very few annotations have been used to mark constructors, fields and parameters.

Finally, another interesting finding was that although Java does not support annotations for anonymous classes, many programs are using it. The reason why the annotation count differ from the `apt` and the textual search is mainly because of those anonymous class annotations. Analyzing the table we can see that in 38 out of 50 systems evaluated by both approches we have anonymous class annotations.

Investigating this occurence further, we discovered another reason for such form of use. Some programming IDEs warn the programmer to use annotations or even generates code with annotations, regardless they are located in anonymous class. We tested this feature using the `Eclipse` IDE build ID 20110218-0911 to generate the code for an anonymous inner class for a button event, and the IDE generated the method with an `@Override` annotation.

V. THREATS TO VALIDITY

Two main factors can impact on the validity of the results presented in this paper. First, our chosen systems might not be representative of all annotations used by the entire Java community. To minimize this threat we used the Qualitas Corpus that has 106 open-source systems from which 65 had annotations.

The second threat is related to the `apt` bugs. We were not able to determine what caused the bug that made the `apt` to crash while processing 15 systems from the Qualitas Corpus. To minimize this threat we developed a second approach based on a textual search. In this way the results obtained by the `apt` could be compared and validated to the results obtained by the textual search. However, using the textual search approach it was not possible to extract the annotations' full qualified name, nor which Java program element have been annotated.

VI. RELATED WORK

Annotations can be considered a relative new feature in the Java language. Probably because of that, there is not many works related to studying the use of annotations in existing systems. However, there are studies related to creating tools to verify the correct use of frameworks and APIs annotations. One of those efforts presents the AnnaBot tool [1], that demonstrate its usefulness using Java Persistence API examples. Another proposed tool to verify frameworks and

APIs annotations is the ModelAn [6], which is a model based approach, and uses the Fraclet as a case study.

On the field of empirical studies about Java constructions and abstractions, there are many related work. First, there is an empirical study of UpgradeJ [10] which is a variant of the Java language. UpgradeJ is a language that allows multiple versions of classes to co-exist, thus supporting dynamic software updatings. The empirical study has evaluated different versions of classes on open-source Java applications and estimated how many of the classes changes could be handle by UpgradeJ dynamically. The results show that most changes could be supported without a significant code rewrite.

There is also a work aiming to evaluate the use of non-private fields in Java applications since there is little empirical evidence to this practice [8]. This study has relied on 100 open-source Java applications (that we could consider to be an ancestor to the Qualitas Corpus [9]). The study has concluded that it is not uncommon for systems to declare non-private fields and do not take advantage of that access.

Another empirical study that uses a Qualitas Corpus release aims to find how programmers are overriding methods [7]. This work relies on a set of metrics to analyze the systems and how well their overriding implementation is. This study found that most subclasses override at least one method. It also found some questionable uses of overriding in the Qualitas Corpus systems.

The analysis of class coupling and the choice of refactoring (or removing) them in latter releases of the systems is the topic of another empirical study [5]. This study shows that the size of the source code does not affect the redesign choices, and a strong tendency that some classes have towards their removal.

Annotations have also been investigated in the scope of aspect-oriented software development. For example, annotations are often mentioned as an alternative to design more robust pointcuts. Kiczales and Mezini recommend using annotations when: (i) it is difficult to write a stable regular expression or enumeration-based pointcut; (ii) the name of the annotation is unlikely to change; (iii) the annotation denotes a well-defined semantic property (and not properties that are only true in some configurations of the system) [4]. Eaddy and Aho propose using annotations at the statement level for exposing join points needed by heterogeneous concerns and for enabling fine-grained advising [2]. However, their proposal can lead to a widespread use of annotation and thus can increase the code scattering and tangling phenomenon usually associated to Java annotations.

VII. Conclusions

We have presented a large empirical study on the use of annotations in open-source Java systems. Our motivation for this work was mainly because as a new language feature, it is difficult to find studies analyzing the use annotations. After analyzing the systems using the official Java annotation processing tool, we had to develop a textual search program to process the systems where the `apt` hast failed. Using the `apt` we were able to process 50 systems with 56,330 annotations; and using the textual search we processed 65 systems with 160,570 annotations.

We have also shown that some systems have a very high annotation density. Moreover, some of the discoveries found by our analysis from both approaches confirm that the most used annotations are the compiler ones. Also that methods are the most annotate element, and that the `JPA` annotations are popular. We also found that programmers are using the `@Test` as a standard annotation to mark methods that will be dynamically tested by reflection.

Finally, although the Java language does not support annotations inside anonymous classes, programmers are still annotating them. Some programming IDEs generate code with annotations even in anonymous classes. This indicates a tendency that maybe should be included on latter versions of the Java language.

As future work we hope to inspect even further the annotations on Java applications, processing and analyzing more annotated data.

References

[1] I. Darwin. Annabot: a static verifier for Java annotation usage. In *2nd International Workshop on Defects in Large Software Systems*, pages 21–28, 2009.

[2] M. Eaddy and A. V. Aho. Statement annotations for fine-grained advising. In *ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, pages 89–99, 2006.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 3rd Edition*. Addison-Wesley, 2005.

[4] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[5] A. Mubarak, S. Counsell, and R. M. Hierons. An empirical study of "removed" classes in Java open-source systems. In *Advanced Techniques in Computing Sciences and Software Engineering*, pages 99–104. Springer, 2010.

[6] C. Noguera and L. Duchien. Annotation framework validation using domain models. In *Model Driven Architecture Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

[7] C. S. Tempero, E. and J. Noble. An empirical study of overriding in open source Java. In *33rd Australasian Computer Science Conference (ACSC)*, volume 102, pages 3–12, 2010.

[8] E. Tempero. How fields are used in Java: An empirical study. *Software Engineering Conference, Australian*, 0:91–100, 2009.

[9] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *Asia Pacific Software Engineering Conference (APSEC)*, pages 336–345, dec 2010.

[10] E. Tempero, G. Bierman, J. Noble, and M. Parkinson. From Java to UpgradeJ: an empirical study. In *1st International Workshop on Hot Topics in Software Upgrades*, pages 1–5, 2008.