Are you still smelling it?

A comparative study between Java and Kotlin language.

Matheus Flauzino
Federal University of Lavras
Lavras-MG, Brazil
matheus.flauzino@posgrad.ufla.br

Júlio Veríssimo
Federal University of Lavras
Lavras-MG, Brazil
julio.santos@posgrad.ufla.br

Ricardo Terra
Federal University of Lavras
Lavras-MG, Brazil
terra@dcc.ufla.br

Elder Cirilo
Federal University of São João del Rei
São João del Rei-MG, Brazil
elder@ufsj.edu.br

Vinicius H. S. Durelli
Federal University of São João del Rei
São João del Rei-MG, Brazil
durelli@ufsj.edu.br

Rafael S. Durelli
Federal University of Lavras
Lavras-MG, Brazil
rafael.durelli@dcc.ufla.br

ABSTRACT

Java is one of the most widely used programming languages. However, Java is a verbose language, thus one of the main drawbacks of the language is that even simple tasks often entail writing a significant amount of code. In some cases, writing too much code might lead to certain code smells, which are violations of fundamental design that can negatively impact the overall quality of programs. To allow programmers to write concise code, JetBrains created a new language named Kotlin. Nevertheless, few studies have evaluated whether Kotlin leads to concise and clearer code in comparison to Java. We conjecture that due to Java's verbosity, programs written in Java are likely to have more code smells than Kotlin programs. Therefore, we set out to evaluate whether some types of code smells are more common in Java programs. To this end, we carried out a large-scale empirical study involving more than 6 million lines of code from programs available in 100 repositories. We found that on average Kotlin programs have less code smells than Java programs.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Code Smell, Bad Smell, refactoring, Kotlin Language

ACM Reference Format:
Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius H. S. Durelli, and Rafael S. Durelli. 2018. Are you still smelling it?: A comparative study between Java and Kotlin language.. In *XII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '18), September 17–21, 2018, Sao Carlos, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3267183.3267186

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SBCARS '18, September 17–21, 2018, Sao Carlos, Brazil
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6554-3/18/09…$15.00
https://doi.org/10.1145/3267183.3267186

1 INTRODUCTION

It is a widely accepted fact among the developer community whose maintenance activities take up a large portion of software development time. code smells (also known as "code smells" or just "smells") compromise the evolution and maintenance of a software system, which can slow down development or increase the propensity for immediate or future failure [7, 39]. Smells are indicators of quality problems that make a software hard to maintain and evolve. Given the importance of smells in the source code's maintainability, many studies have explored the characteristics of smells and analyzed their effects on the software quality. Another fact is related to the growing number of studies that present techniques and tools for detecting smells in the most varied contexts [3, 19, 27].

The majority of the research on code smells is geared towards Java source code. Undoubtedly, Java is an extremely popular language, ranking first for a long time in the TIOBE[1] Programming Community Index. However, there are several JVM-based languages that have Java-like evolutionary features, such as Groovy, Scala, JRuby, Jython, Kotlin, etc.

Lots of inspiration was drawn from these JVM-based language during the development of Kotlin. Kotlin has been devised by JetBrains to help programmers write concise and crisp code to help save ample time and decrease the clustering and boilerplate code. According to JetBrains, Kotlin understands the code and can infer the type of variable declaration as well as getters/equals/hashcode generated by the compiler. It helps the programmer to get rid of hassling task. Therefore Kotlin aims to save time as well as helps in increasing productivity. Kotlin's syntax focuses on removing verbosity, i.e., rough estimates indicate approximately a 40% cut in the number of LOC [28]. Kotlin is concise, safe, and focused on interoperability with Java code. It can be used almost everywhere Java is used today: for server-side development, Android apps, and much more. Kotlin works great with all existing Java libraries and frameworks, and runs with the same level of performance as Java. Just like Java, Kotlin is a statically typed programming language. This means the type of every expression in a program is known at compile time. Ever since Google announce Kotlin as the official language many repositories in GitHub seems to migrate to Kotlin.

As stated before, nowadays it is possible to find a set of research based on code smells for different domain and perspectives.

[1] https://www.tiobe.com/tiobe-index/

There are diverse types of tools to detect code smells such as PMD, JDeodorant [31, 32], Checkstyle, etc. These tools use alternatives approaches, graph-based techniques [33], mining of code changes [23], metrics-based detection [14–16], textual-based technique [24], using machine learning techniques to detect smells [18], vector machine learning technique [9], etc. There are also recent researches investigating how relevant code smells are for programmers [20, 41] and how code smells evolve over time [34]. However, to the best of our knowledge there is none research comparing Java and Kotlin regarding code smells. Dealing with the aforementioned issue, our main motivation is to find out if Kotlin contains less code smells when compared to Java. Note that, in the context of this paper, verbosity means how much code developers can see and parser in a single glance—"developers need to use too many words making the code noise". On the other hand, excessive terseness in a language also can cost mental energy [29].

We present details of our approach in Section 4. Summing up, we **investigated more than 6 million of lines on code (LOC) in 100 open-source repositories (50 Kotlin/50 Java), selected by order of popularity (stargazers count) in GitHub**[2]. After choosing the repositories, we downloaded the source codes and executed smells analysis tools separately for each language and we focused in five well-known and common smells. Although Kotlin is a relatively new language, it was created with the intention of being used in conjunction with the Java language. For example, Kotlin allows migration from Java to Kotlin and vice-versa. That way, 100% compatible codes and libraries between Java and Kotlin can be used. In addition, according to GitHub, nowadays, there are approximately 27,797 projects that are already using Kotlin. Thus, for our experiment, we tried to be as unbiased as possible when selecting repositories. For instance, during the "stargazers count" filter, we tried to balance projects in domains such as desktop and mobile.

The contributions of this work are fourfold:

(1) **A large-scale empirical study involving 100 open-source repositories and two programming languages (Kotlin and Java)**, aiming to report quantitative and qualitative evidence on which language presents less code smells.

(2) **The first study that evaluates the incidence of code smell in the Kotlin language** to our knowledge;

(3) This is also the **first study comparing code smells incidences between the Java and Kotlin languages**;

(4) **A publicly available dataset**[3] to allow other researchers to conduct similar studies on code smells and the relationship between the two languages (or simply reproduce our results).

Paper Structure. Section 2 presents the concept of code smells and we detail the types of smells analyzed in this paper. We also present in Section 3 a brief comparison of the Java and Kotlin languages. Sections 4 and 5 address the experiment setup and the experiment results, respectively. Section 6 presents the discussion. Section 7 presents the related works. Section 8 describes the concluding remarks, lessons learned, and future work.

[2] https://github.com

[3] https://github.com/matheusflauzino/smells-experiment-Kotlin-and-Java

2 CODE SMELLS

Code smells are clusters of negative decisions made by developers in software design that do not directly affect the execution flow of a program, but are potential causes of future problems, increasing the complexity, maintainability, and even the cost of software [4]. Fowler et al. [4] described 22 smells and incorporated them into refactoring strategies to improve design quality, they highlighted the fundamental role of human judgment, but in the last few years detection algorithms have emerged automated [14]. A simple example of metric-based detection is the *Long Parameter List* smell, whose unique metric is the Number Of Parameters (NOP). Other detection strategies use more concepts and explore the more extensive metrics, combining several metrics and logical expressions in order to obtain specific characteristics of code smells with greater precision [11, 12, 17].

In addition to the *Long Parameter List*, there are several many other smells and always increasing [1]. This paper, nevertheless, focuses on the following:**Data Class:** This smell refers to a class that contains only raw fields and methods to access them, usually consisting of *getters* and *setters* without any behavior (methods). It is nothing more than a container for data used by other classes. **Large Class:** Classes that are trying to do too much often have large numbers of instance variables. Sometimes groups of variables can be clumped together. Sometimes they are only used occasionally. Over-large classes can also suffer from code duplication. **Long Method:** It is a method that contains too many lines of code. They are bad because long methods are hard to understand. Long methods should be decomposed for clarity and ease of maintenance. **Long Parameter List:** It occurs when methods contain an excessive number of parameters. They are bad because they are difficult to understand and use, and can become easily inconsistent. You do not need to pass in everything a method needs, just enough so it can find all it needs. **Too Many Methods:** Classes that have many methods violating the principle of responsibility. It would be preferable to extract functionalities that clearly belong together in separate parts of the code.

We selected this group of smells because, in addition to being the most reported in current researches [29]. Furthermore, another important factor that led us to choose the aforementioned code smells is that these five smells are very common smells for both PMD and Detekt. Therefore, we claim that they allows us to draw a comparison between Java and Kotlin programs.

3 JAVA VERSUS KOTLIN LANGUAGE

Java is an extremely popular and widely used language. It has been designed with the goal of having platform independence, i.e., having portability, working with network resources, and being secure.

Kotlin is a concise language, with estimates of being less verbose than Java. Although it is a new language (the project started in 2010, with the first version launched in 2016), the language is behind big companies such as JetBrains (by developer) and Google, which has joined as recommended language for Android (because it is 100% compatible with Java). Kotlin shows greater security (i.e., support for non-nullable types makes applications less prone to null point dereference (a.k.a., *NullPointerException*). It also includes smart casting, higher-order functions and extension functions.

To understand a little more this comparison between the two languages, we show two snippets of code comparing the two languages, where we implement a declaration of a Book class with the attributes `title` and `author` and their accessor methods (getters and setters). Listings 1 and 2 demonstrate the respective implementation in Java and Kotlin. We can note a large difference in the number of LOC for the same implementation, Java requires 16 lines whereas Kotlin just one.

```
1  class Book {
2     private String title;
3     private Author author;
4     public String getTitle() {
5         return title;
6     }
7     public void setTitle(String title) {
8         this.title = title;
9     }
10    public Author getAuthor() {
11        return author;
12    }
13    public void setAuthor(Author author) {
14        this.author = author;
15    }
16 }
```

Listing 1: Java example

```
1  data class Book(var title:String,var aut:Author)
```

Listing 2: Kotlin example

4 EXPERIMENT SETUP

In this section, we describe our experiment. We set out to investigate which programming language contains more code smells: Java or Kotlin? We conjecture that Kotlin contains less bad smell once it has been created to be a programming language where one writes less source code.

We pose the following research question (**RQ**): Do Java and Kotlin differ in the occurrence of code smells such as: Data Class, Large Class, Long Method, Long Parameter List, and Too Many Methods?

4.1 Scope

Defining the scope of an experiment comes down to setting its goals. We used to organization proposed by the Goal/Question/Metric (GQM) [38] template to do so. Therefore, the scope of this study can be summed up as follows:

Analyze whether Kotlin contains less bad smell when compared to Java
For the purpose of evaluation
With respect to reusability and maintainability
From the point of view of the researcher
In the context of software engineering.

4.2 Hypotheses Formulation

The **RQ** was translated into the following hypotheses:

Null Hypothesis, H_{0DC}: there is no difference between bad smell **Data Class** in Kotlin and in Java.

Alternative Hypothesis, H_{1DC}: there is a significant difference between bad smell **Data Class** in Kotlin and in Java.

Then, the hypotheses can be formally stated as:

$$H_{0DC}: \mu_{javaDC} = \mu_{kotlinDC}$$

and

$$H_{1DC}: \mu_{javaDC} > \mu_{kotlinDC}$$

Null Hypothesis, H_{0LC}: there is no difference between bad smell **Large Class** in Kotlin and in Java.

Alternative Hypothesis, H_{1LC}: there is a significant difference between bad smell **Large Class** in Kotlin and in Java.

$$H_{0LC}: \mu_{javaLC} = \mu_{kotlinLC}$$

and

$$H_{1LC}: \mu_{javaLC} > \mu_{kotlinLC}$$

Null Hypothesis, H_{0LM}: there is no difference between bad smell **Long Method** in Kotlin and in Java.

Alternative Hypothesis, H_{1LM}: there is a significant difference between bad smell **Long Method** in Kotlin and in Java.

$$H_{0LM}: \mu_{javaLM} = \mu_{kotlinLM}$$

and

$$H_{1LM}: \mu_{javaLM} > \mu_{kotlinLM}$$

Null Hypothesis, H_{0LPL}: there is no difference between bad smell **Long Parameter List** in Kotlin and in Java.

Alternative Hypothesis, H_{1LPL}: there is a significant difference between bad smell **Long Parameter List** in Kotlin and in Java.

$$H_{0LPL}: \mu_{javaLPL} = \mu_{kotlinLPL}$$

and

$$H_{1LPL}: \mu_{javaLPL} > \mu_{kotlinLPL}$$

Null Hypothesis, H_{0TMM}: there is no difference between bad smell **Too Many Methods** in Kotlin and in Java.

Alternative Hypothesis, H_{1TMM}: there is a significant difference between bad smell **Too Many Methods** in Kotlin and in Java.

$$\mathbf{H}_{0TMM}: \mu_{javaTMM} = \mu_{kotlinTMM}$$

and

$$\mathbf{H}_{1TMM}: \mu_{javaTMM} > \mu_{kotlinTMM}$$

Our main goal is to investigate which programming language contains less code smells. Therefore, this experiment has one factor (number of code smells) and two treatments (Java and Kotlin). Let μ the average of code smells, thus μ_{java} and μ_{kotlin} denote the averages of code smells in Java and in Kotlin, respectively.

4.3 Variables Selection

As mentioned, the purpose of this experiment is to evaluate whether Kotlin contains less code smells than Java. Thus, we are particularly interested in the following dependent variable—identified code smells—and the three following independent variables: (*i*) results from PMD (Java); (*ii*) results from Detekt (Kotlin); and (*iii*) open-source systems as reported in the first columns of Tables 1 and 2.

PMD and Detekt are static code analysis tools for Java and Kotlin, respectively. Herein we have used the latest version currently available for both tools. PMD uses styles rules for the identification of smells. Similarly, the authors of Detekt performed an extension of the PMD. This extension is strongly based on the same styles rules.

4.4 Sample Selection

To perform this experiment, we divided the population of open-source systems from GitHub into two groups: Java projects and Kotlin projects. We tried to include a wide range of systems that differ in size, complexity, and category.

We selected 100 open-source systems from GitHub—50 Java and 50 Kotlin. We have selected and downloaded the top 50 Java and the top 50 Kotlin repositories ordered by popularity (stargazers count) in GitHub. We followed the guidelines proposed by Kalliamvakou et al. [8] during the construction of our sample.

4.5 Operation

First, we selected the top 100 repositories[3]—50 Java and 50 Kotlin—ordered by popularity in GitHub (stargazers count). From this initial list, we discarded the lower quartile ordered by number of commits, to focus the study on repositories with more maintenance activity. The final selection consists of well-known projects, such as: spring-boot, RxJava, and JetBrains/kotlin-native. As shown in Tables 1 and 2, the size of the projects ranges from 131 to 1,055,917 lines of code. On average, the projects contain around 61,877 lines of code. More details about the lines of code can be seen in Figure 1.

Figure 2 shows line plots with the distribution of the of number of commits. It can be observed that some projects have the amount of commits relatively similar. However, it can also be observed that Java contains more commits when compared to Kotlin. We believe that this happens since Kotlin is relatively a new language when compared to Java. The same holds when comparing the project's lifespan. Figure 3 describes the project's lifespan (in days).

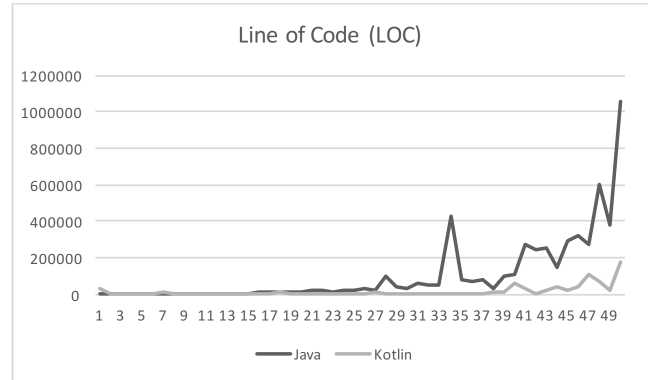

Figure 1: Distribution of project's LOC.

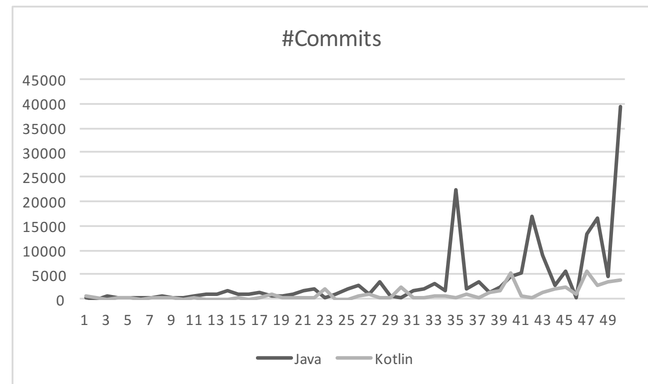

Figure 2: Distribution of project's #Commits.

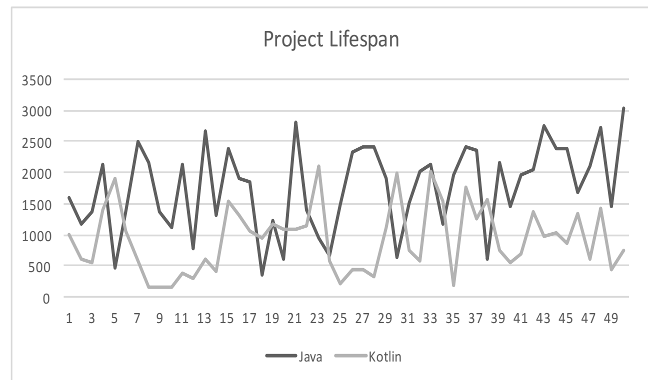

Figure 3: Distribution of project's lifespan.

Table 1: Java projects we analyzed in our experiment. The entries in the table are in ascending order by the total of identified bad smells.

Program Name	Bad Smell DC	LC	LM	LPL	TMM	Total	#Commits	LOC	Lifespan
CircleImageView	0	0	0	0	0	0	122	356	1593
Material-Animations	0	0	0	0	1	1	76	992	1178
RxAndroid	0	0	0	0	1	1	461	1181	1379
PhotoView	0	0	1	0	0	1	427	1589	2142
interviews	0	0	1	0	0	1	390	7300	469
AndroidSwipeLayout	0	1	1	0	1	3	175	2667	1373
ViewPagerIndicator	0	0	1	0	3	4	241	3072	2490
SlidingMenu	0	0	2	0	3	5	723	3212	2159
Android-CleanArchitecture	3	0	0	0	3	6	243	2777	1379
leakcanary	2	0	1	0	5	8	454	4243	1126
EventBus	2	0	0	0	10	12	480	5333	2143
BaseRecyclerViewAdapterHelper	7	1	1	0	3	12	908	5672	779
android-async-http	3	1	3	0	7	14	874	3446	2655
material-dialogs	0	2	4	0	8	14	1560	7584	1303
Android-Universal-Image-Loader	3	1	1	0	11	16	1025	5276	2375
butterknife	0	2	8	1	7	18	836	10653	1911
picasso	0	1	1	2	16	20	1198	9632	1841
SmartRefreshLayout	1	1	9	2	10	23	723	17064	361
stetho	2	0	2	0	19	23	528	18214	1223
lottie-android	11	0	3	4	11	29	925	11214	600
retrofit	1	3	2	0	23	29	1572	19419	2822
java-design-patterns	31	0	1	0	5	37	2060	27426	1389
plaid	12	1	9	6	16	44	453	16921	942
AndroidUtilCode	6	8	5	2	28	49	863	22481	668
MPAndroidChart	5	3	25	2	15	50	1983	24910	1495
androidannotations	16	0	5	1	45	67	2774	37211	2335
greenDAO	30	0	0	5	33	68	844	19663	2411
zxing	18	0	21	9	26	74	3431	105107	2421
jadx	22	1	9	0	46	78	662	44942	1898
tinker	8	2	35	16	26	87	295	31826	630
iosched	18	9	25	2	53	107	1548	57854	1519
Hystrix	17	5	23	23	44	112	2106	50510	2017
okhttp	7	11	6	4	88	116	3159	56228	2136
fresco	6	1	5	10	98	120	1657	428112	1184
selenium	8	2	8	2	110	130	22349	85807	1961
glide	1	3	7	9	117	137	2211	70355	2411
Signal-Android	25	9	10	17	79	140	3543	80634	2357
zheng	75	3	4	0	76	158	1222	30057	602
incubator-dubbo	89	11	40	4	146	290	2308	103349	2170
ExoPlayer	48	19	35	58	149	309	4788	113451	1446
RxJava	0	54	55	8	364	481	5366	271036	1967
spring-boot	276	10	2	2	245	535	16829	247388	2048
netty	22	35	84	26	485	652	8797	250399	2758
fastjson	490	19	89	5	80	683	2705	149361	2399
druid	297	44	209	1	176	727	5812	295170	2399
Telegram	54	72	523	62	192	903	311	325744	1677
libgdx	70	37	141	266	462	976	13254	274702	2118
spring-framework	362	53	81	17	818	1331	16597	602324	2729
guava	10	82	49	34	1167	1342	4725	381338	1461
elasticsearch	351	86	415	214	1266	2332	39203	1055917	3032

4.6 Execution

We have created an script to download these 100 repositories. After that, we needed to apply tools to identify the code smells in these 100 repositories. Herein we have used two tools: (i) PMD and (ii) Detekt. We have created two more scripts - one to execute PMD and another one to execute Detekt. Herein, we analyzed five well-known code smells detected by these tools. The analyzed code smells are: (*i*) Data Class (DC), (*ii*) Large Class (LC), (*iii*) Long Method (LM), (*iv*) Long Parameter List (LPL), and (*v*) Too Many Methods (TMM). As result, we have get all identified code smells in these repositories. We also have created a script to count the line of codes. All information can be seen in Tables 1 and 2.

5 EXPERIMENTAL RESULTS

In this section we first discuss some descriptive statistics (see Subsection 5.1), present the hypothesis testing for the 100 projects we analyzed(see Subsection 5.2) , and then we present the threats to validity in subsection 5.3.

5.1 Descriptive Statistics

As shown in Tables 1 and 2, Java has presented more code smells than Kotlin. `Elasticsearch` is the Java project that contains more code smells. This project contains 39,203 commits and its lifespan is 3,032 days. Table 1 reports that PMD could not find any bad smell to project `CircleImageView`. We argue that this is a reflection of

Table 2: Kotlin projects we analyzed in our experiment. The entries in the table are in ascending order by the total of identified bad smells.

Program Name	Bad Smell DC	LC	LM	LPL	TMM	Total	#Commits	LOC	Lifespan
awesome-kotlin	0	0	0	0	0	0	814	35825	1016
Design-Patterns-In-Kotlin	0	0	0	0	0	0	101	405	610
GankClient-Kotlin	0	0	0	0	0	0	46	894	553
gradle-play-publisher	0	0	0	0	0	0	357	915	1391
kotlin-examples	0	0	0	0	0	0	154	1545	1898
Kotlin-for-Android-Developers	0	0	0	0	0	0	22	685	1057
Kotlin-Tutorials	0	0	0	0	0	0	118	10398	597
profile-summary-for-github	0	0	0	0	0	0	194	350	170
SdkSearch	0	0	0	0	0	0	450	2488	166
transitioner	0	0	0	0	0	0	33	131	172
android-architecture-components	0	0	0	0	1	1	168	6174	385
android-clean-architecture-boilerplate	0	0	0	0	1	1	65	2181	293
Colorful	0	0	0	1	0	1	53	264	602
JellyToolbar	0	0	0	0	1	1	20	425	427
kotlin-koans	0	0	0	0	1	1	214	1696	1531
kotterknife	0	0	0	0	1	1	82	251	1317
Android-TextView-LinkBuilder	0	0	0	0	2	2	111	703	1070
sqldelight	0	0	0	0	2	2	860	11024	951
Bandhook-Kotlin	0	0	0	1	2	3	107	3167	1165
dexcount-gradle-plugin	0	2	1	0	1	4	300	1177	1090
Fuel	0	0	0	0	5	5	423	5581	1090
TourGuide	0	1	0	0	4	5	190	1760	1152
mapdb	0	0	0	0	6	6	2100	2190	2111
SearchFilter	0	2	1	0	3	6	23	1381	579
kotlinconf-app	0	1	0	0	6	7	6	3879	215
Fotoapparat	0	0	0	0	9	9	582	5703	432
kotlinpoet	0	2	0	1	6	9	1111	10398	454
p3c	0	0	0	4	5	9	172	5302	340
android-topeka	0	1	0	0	9	10	239	4077	1117
shadowsocks-android	0	1	2	1	8	12	2412	5354	1990
flexbox-layout	0	2	9	1	2	14	329	8206	755
RxDownload	0	1	0	0	13	14	456	2723	578
spek	0	0	0	5	11	16	586	3819	2031
okio	0	4	0	1	12	17	663	5733	1534
android-ktx	0	2	2	0	14	18	438	5894	182
Exposed	0	2	0	2	14	18	951	8333	1764
SuperSLiM	0	3	0	11	7	21	177	4338	1258
muzei	0	5	3	3	13	24	1496	12153	1573
kotlin-dsl	0	0	0	8	17	25	1902	15207	762
intellij-rust	0	5	0	1	21	27	5495	60348	553
kotlinx.coroutines	0	6	1	0	21	28	804	31567	706
RxKotlin	0	0	0	30	5	35	211	1850	1379
tachiyomi	0	9	1	2	58	70	1417	24902	965
ktor	0	5	1	32	34	72	2013	38172	1030
tornadofx	0	9	0	37	37	83	2535	21082	874
anko	0	1	0	8	125	134	994	38169	1349
corda	0	27	13	66	88	194	5814	106854	600
Twidere-Android	0	37	45	66	94	242	2888	69499	1426
arrow	0	0	0	294	51	345	3672	22829	431
kotlin-native	0	55	200	251	208	714	3771	178650	743

the few LOC (356) that have been evaluated in this project. It also contains just 122 commits.

Table 3 illustrates descriptive statistics of all code smells identified in Java projects in the 50 repositories. On one hand, the bad smell that has been most identified was TMM (6,597). On the other hand, the one that has been less identified was LC (593). Regarding the others, DC was identified 2,409, LM 1,962, and LPL 814 times.

As can be seen in Table 3, the mean (μ) of all identified code smells are DC = 7, LC = 1.5, LM = 5, LPL = 2, and TMM = 26. Due to the outliers in our data concerning the amount of code smells identified by PMD in Java projects, we consider the trimmed mean and the median values in Table 3 to be more accurate measures of central tendency than the mean. The purpose is to calculate a mean that represents most of the values well and is not unduly influenced by extreme values.

Table 3: Descriptive statistics summarizing the characteristics of all code smells identified in Java projects.

	DC	LC	LM	LPL	TMM
Total	2,409	593	1,962	814	6,597
Mean (μ)	7	1.5	5	2	26
Average	48.18	11.86	39.24	16.28	131.94
Trimmed	15.825	6.15	14.625	4.5	59.925
STDEV	108.016	22.086	97.548	48.275	270.746
MAD	7	1.5	4	2	24

Additionally, also due to aforementioned outliers, the median absolute deviation (MAD) is a more robust measure of statistical dispersion than the standard deviation (refer to Table 3). Figure 4 shows a set of box-plot with the distribution of all code smells identified in Java and Kotlin projects. We provide plot for all the columns DC, LC, LM, LPL, and TMM in Tables 1 and 2.

As the age of a project can be an indicator of project maturity, we also looked into the lifespan of the projects. As can be seen in Table 1, the most mature project is `elasticsearch` and—as stated earlier—it is the project with more code smells, more LOC, and more commits. This may indicate that the more commits are performed, the more code smells are added [22, 23], i.e., smells are generally the result of continuous maintenance activities. On the other hand, the project that has the lowest level of maturity is the `SmartRefreshLayout`, which we have identified 23 code smells.

The Kotlin project that contains more code smells is `kotlin-native`. It contains 3,771 commits and its lifespan is 743 days. Detekt could not find any code smells in ten projects (refer to Table 2). We believe this is due to their LOC, e.g., project `transitioner` contains only 131 LOC and only 33 commits, and project `Kotlin-forAndroid-Developers` contains just 685 LOC and 22 commits.

Table 6 illustrates descriptive statistics of all code smells identified in Kotlin projects. Similar to the Java language, the most identified bad smell was TMM (918). However, as can also be noted in Table 6, Detekt could not find any bad smell DC in the 50 Kotlin projects. We believe that this is because DC refers to a class that contains only fields and accessor methods. As illustrated in Listing 2, Kotlin does not need to syntactically create such methods, i.e., getters and setters. The second most-identified bad smell was LPL with a total of 826, followed by LM (279) and LC (183).

Also due to the outliers in our data in the Kotlin projects, we consider the trimmed mean and the median values in Table 6 to be more accurate. We also measure the MAD for Kotlin data. Figure 4 also contains the distribution of identified code smells in Kotlin projects.

The most mature Kotlin project in this experiment is `mapdb` and Detekt was able to identify an amount of six code smells (TMM). `Corda` is the project that most contains maintenance activities (5,814 commits). However, this is quite a surprising findings, considering that it contains only 194 code smells and its lifespan is 600 days. On the other hand, `kotlinconf-app` owns only six commits and a total of seven code smells.

5.2 Test of the Hypotheses

After data collection, we applied statistical tests to the experiment results. As reported in Table 4, we checked whether or not our data follow a normal distribution applying the Shapiro-Wilk test. As every *p-value* was greater than 0.05, it can be stated, with a confidence level of 95%, that all data follow a normal distribution.

Given that our data follow a normal distribution, we applied the paired t-test to verify our hypotheses. As shown in Table 5, all code smells had *p-values* < 0.05. Therefore, we could reject four null hypotheses and therefore we could find significant results, i.e., we conclude that there is a statistically significant difference between the four code smells when comparing Java and Kotlin languages. Note that except for LPL, we can statistically conclude that Kotlin contains less code smells than Java.

Summary for RQ. In general, Kotlin contains less code smells when compared to Java according to descriptive statistics. Since all data follow a normal distribution, results of paired t-tests assign to Java the higher number for four out of five code smells analyzed in this paper.

5.3 Threats to Validity

In this section, we consider the study of [6] as a template to discuss the threats that might jeopardize the validity of our experiment. Internal validity is concerned with the confidence that can be placed in the cause-effect relationship between the treatments and the dependent variables in the experiment. External validity has to do with generalization, namely, whether or not the cause-effect relationship between the treatments and the dependent variables can be generalized outside the scope of the experiment. Conclusion validity focuses on the conclusions that can be drawn from the relationship between treatment and outcome. Finally, construct validity is about the adequacy of the treatments in reflecting the cause and the suitability of the outcomes in representing the effect. We categorized all threats to validity according to this classification.

5.3.1 Internal Validity. We mitigated the selection bias issue by using randomization. However, since we assumed that all types of projects have the same characteristics, no blocking factor was applied to minimize the threat of possible variations in, for instance, the complexity of the projects. Thus, we cannot rule out the possibility that some variability in how end users perceive the quality of the chosen project stems from other quality factors as opposed to the amount of fault-related problems in the chosen project.

5.3.2 External Validity. The sample might not be representative of the target population. As mentioned, we randomly selected projects for the treatment. However, we cannot rule out the threat that the results could have been different if another sample had been selected. Another potential threat to the external validity of our results is that our analysis includes only projects from GitHub. Although, GitHub is a web-based hosting service for version control using git, further investigation involving more project from another git hosting service is required before we can determine whether our findings can be generalized for different projects and different developer populations. Consequently, we cannot be confident that the results hold for a more representative sample of the general population. Yet, in spite of that, we believe that the insights gained from this study can be generalized to similar settings.

5.3.3 Conclusion Validity. The main threat to conclusion validity has to do with the quality of the data collected during the course of the experiment. More specifically, the quality of code smells is of key importance to the interpretation of our results. Given that all code smells have been identified by third tools that might not always reflect the true amount code smells (false negative and false positive). Thus, it is possible that some data is not correct due to

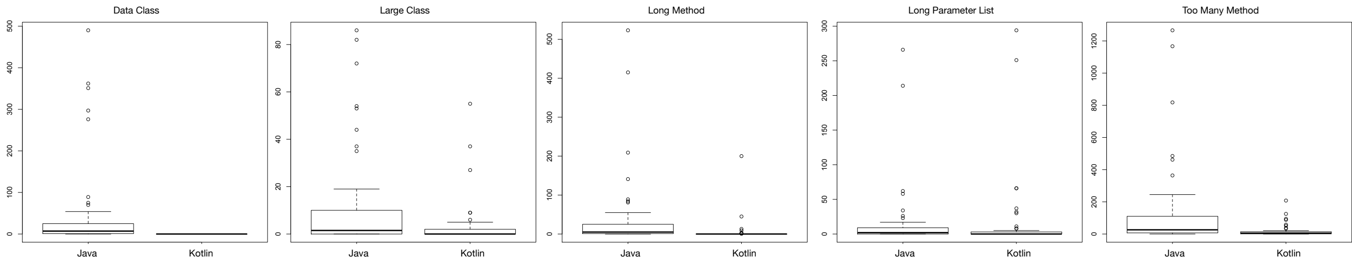

Figure 4: Distribution of the code smells in Java and Kotlin.

Table 4: Results of the Shapiro-Wilk test.

Data Class		Large Class		Long Method		Long Parameter List		Too Many Methods	
w	p-value	w	p-value	w	p-value	w	p-value	w	p-value
0.33372	2.20E-16	0.4998	2.20E-16	0.33292	2.20E-16	0.34484	2.20E-16	0.3973	2.20E-16

Table 5: Results of the Paired T-Test.

code smells	T	p-value	Mean of the diff.	Result
Data Class	3.154	0.001375	48.18	H_{0DC} is refuted due less significance than 5%
Large Class	2.7114	0.004606	8.2	H_{0LC} is refuted due less significance than 5%
Long Method	2.4605	0.008723	33.66	H_{0LM} is refuted due less significance than 5%
Long Parameter List	-0.0337	0.5134	-0.24	H_{0LPL} is refuted due less significance than 5%
Too Many Methods	3.0443	0.001873	113.58	H_{0TMM} is refuted due less significance than 5%

Table 6: Descriptive statistics summarizing the characteristics of all code smells identified in Kotlin projects.

	DC	LC	LM	LPL	TMM
Total	0	183	279	826	918
Mean (μ)	0	0	0	0	5
Average	0	3.66	5.58	16.52	18.36
Trimmed	0	1.15	0.225	2.8	8.625
STDEV	0	9.878	28.834	54.978	37.735
MAD	0	0	0	0	5

misguided or ill-identified code smells or even fabricated information. It is worth noting, however, that data inconsistencies were filtered out during the experiment.

5.3.4 Construct Validity. The measures used in the experiment may not be appropriate to quantify the effects we intend to investigate. If the measures we used do not properly match the target theoretical construct, the results of the experiment might be less reliable.

6 DISCUSSION

Results show that Kotlin tend to contain less code smells when compared to Java. We claim this happens because usually developers do not like to write tons of lines of code, responsible for most smells. Kotlin tries to help developers write more concise and crisper code to gain time, and decrease the clustering and boilerplate code. Kotlin was devised to understand the code and infer the type of variable as well as getters and setters generated by the compiler. Kotlin also tries to help developers to get rid of tedious tasks in hopes of improving productivity.

As mentioned in Section 3, Kotlin is a relatively new language that brings the better of two paradigms, Object-Oriented (OO) and Functional Programming (FP), i.e., it allows the developer to use OO and FP, or simply to combine them. This also allows developers to focus on making code more readable and less verbose. Kotlin has been designed to minimize many problems that developers have with Java. We believe that these are some of the key factors to support our results, since Kotlin contains less code smells when compared to Java.

We have also seen in this study that the vast majority of the analyzed smells are directly related to the number of lines of code (LOC) [39–42]. We know that Kotlin by nature has a significant reduction when writing these code. For instance, Section 3 brought

up a comparison between Java and Kotlin where it is possible to perceive (see Listing 2) how Kotlin is concise and straightforward to declare a class with its getters and setters. Conversely, we see that we have the same result as the Java example (see Listing 1) with only one single line of code in Kotlin (Listing 2). We know that some smells like DC, LC, and LM naturally have a lower incidence of smells when compared to Java, we can state that these characteristics of Kotlin may be one of the great factors for these smells to have had less incidence when compared with Java. Another fact that supports this assertion is that in the repositories analyzed in Kotlin - see Table 2 - the number of DC was zero.

We also observed that the LPL was the only smell that in Kotlin presented a greater number of incidents than in Java. We believe that this is due to the fact that the parameter number is more related to good practices performed by the developer than a characteristic of the language itself. However, we cannot rule out the fact that the agreement between the detectors is low [2], i.e., different tools can identify the smelliness of different code elements.

7 RELATED WORK

As far as we know, there is no research that directly correlates the presence of code smells with Java and Kotlin. There is a strong growing interest in the use of the Kotlin programming language in scientific and engineering applications [10, 26]. While Java is one of today's most popular languages and also the official language of Android developers, its use is not always the best option [13]. Due to the lack of smells correlations between the two languages, we brought an overview of code smells and smells detection approaches, and how important it is for developers to understand and identify smells in their software.

Many approaches and tools have been proposed to analyze smells in Java [3, 30, 35–37]. However, there are still few tools that detect smells in Kotlin, highlighting Detekt[4] and Ktlint[5].

Walter and Alkhaeir [37] present a study that analyzes separately the impacts of smells and design patterns on the quality of the code, evaluating the correlation with the lack of design pattern and the presence of code smells. The authors analyzed nine design patterns and seven code smells in two open-source Java systems in which they came to the conclusion that classes with design patterns appear to display code smells less often than other classes. Among these seven smell only one (Data Class) was used in our study - once PMD and Detekt tools do not provide style rules for the other six smells.

Similarly, Palomba et al. [22] describe a large-scale empirical investigation on the diffuseness on code smells in open-source Java projects and their impact on code change and susceptibility to failure. The study was conducted in 395 versions on 30 projects, in which it considered 17,350 instances on 13 different types on code smells, doing an analysis based initially on metrics and then validating manually. They show how code smells should be monitored by programmers, since they are related to maintainability aspects such as propensity for changes and failures. Another important factor on this paper is the importance on developing new tools to automate the identification and removal of smells. Herein, we have used 100 projects, 50 Kotlin's projects and 50 Java's projects.

Understanding the useful life of code smells over a software project has also been the focus of some researchers [5, 21]. Peters and Zaidman [25] point out the useful life of code smells in a particular group of software systems. They evaluated seven open-source systems in order to investigate the code smells lifespan and the refactoring behavior of the developers involved in these projects, based on information about the behavior of these developers in relation to the code smells resolution. The authors concluded that the developers involved in the projects are aware of the presence of code smell in the system, but are not very concerned about the impacts that can cause such smells, giving low importance in refactoring activities.

Understanding the impacts of smells regarding maintainability can help to prevent them. According to several studies, smells are heavily influenced by code size, i.e., LOC. In the same way, as carried out in our experiment, we can highlight two empirical studies related to maintenance and LOC. Yamashita and Moonen [42] pointed out the importance of understanding the extent to which maintenance problems influence the prediction of code smells. The authors investigate to which extent problems concerning maintenance can be predicted by the detection of currently known code smells.

Following the same line of study, Yamashita and Counsell [40] carry out an empirical study to investigate whether the code smells are manifestations of design flaws that can degrade code maintainability. The authors evaluated four medium-sized Java systems using code smells and compared the results against previous evaluations on the same systems based on expert judgment. As a result, the authors claim that most code smells are strongly influenced by LOC. In the context of our experiment this seems to proceed (see Section 5). In addition, LOC seems to suggest a major influencer in some smells for Kotlin's language.

8 CONCLUDING REMARKS

We present a large-scale empirical study comparing two languages (Java and Kotlin), statistically evidencing the incidences of code smells between these languages. We investigated 50 Java and 50 Kotlin projects—it was intended to understand if Kotlin contains fewer smells than Java. Our findings support the hypothesis that Kotlin presents less code smells than Java according to descriptive statistics — except for LPL. We validate these results based on the paired t-test, we refuted four null hypotheses with a confidence level of 95%. In addition, we believe that since Kotlin is a more concise and easy-to-understand language developers can create programming artifacts with a significant reduction in the number LOC. This may have been responsible for the result, where we have seen that the smells that are directly bound to this metric (LOC)—such as DC, LC, and LM—presented a considerable difference in Kotlin when compared to Java.

As future work, we plan to analyze and understand the evolution of code smells between Kotlin and Java throughout the life cycle of the repositories, evaluating their commits over time to understand each language, when smells are introduced, why they are introduced, how much time they survive, and how developers

[4]https://arturbosch.github.io/detekt/
[5]https://ktlint.github.io/

remove them. Another future work we plan to discuss in details how Kotlins' features could be useful to reduce a set of smells. In addition, we plan to expand the search for comparisons with other JVM-based language, such as Groovy or Scala, which are languages before Kotlin and hence to understand the incidence of smells in languages that have been designed to be an evolution and less verbose than Java. Finally, we plan to compare JavaScript and Kotlin to better assess and characterize whether Kotlin contains less smells in other domains, such as web.

ACKNOWLEDGMENT

This research is supported by CAPES and FAPEMIG.

REFERENCES

[1] Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac. 2017. Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring. In *4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 205–206.

[2] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* 11, 2 (2012), 5–10.

[3] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä. 2013. Code Smell Detection: Towards a Machine Learning-Based Approach. In *29th International Conference on Software Maintenance (ICSM)*. 396–399.

[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[5] Shizhe Fu and Beijun Shen. 2015. Code Bad Smell Detection through Evolutionary Data Mining. In *9th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–9.

[6] Ray Hyman. 1982. Quasi-Experimentation: Design and Analysis Issues for Field Settings. *Journal of Personality Assessment* 46, 1 (1982), 96–97.

[7] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Foutse Khomh. 2013. Mining the relationship between anti-patterns dependencies and fault-proneness. In *20th Working Conference on Reverse Engineering (WCRE)*. 351–360.

[8] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. 2014. The Promises and Perils of Mining GitHub. In *11th Working Conference on Mining Software Repositories (MSR)*. 92–101.

[9] Amandeep Kaur, Sushma Jain, and Shivani Goel. 2017. A Support Vector Machine Based Approach for Code Smell Detection. In *International Conference on Machine Learning and Data Science (MLDS)*. 9–14.

[10] P V Kulkarni and Mayur K Jadhav. 2018. Exploring Kotlin's Enhancements for multiplatform projects. *International Education and Research Journal* 4, 3 (2018).

[11] Michele Lanza, Radu Marinescu, and Stéphane Ducasse. 2005. Object-Oriented Metrics in Practice. Secaucus.

[12] Jörg Lenhard, Martin Blom, and Sebastian Herold. 2018. Exploring the suitability of source code metrics for indicating architectural inconsistencies. *Software Quality Journal* (March 2018), online.

[13] Pranita Maldikar, San-Hong Li, and Kingsum Chow. 2016. Java Performance Mysteries. *ITM Web of Conferences* 7 (2016), 09015.

[14] Radu Marinescu. 2004. Detection strategies: metrics-based rules for detecting design flaws. In *20th International Conference on Software Maintenance (ICSE)*. 350–359.

[15] B. M. Merzah and Y. E. Selçuk. 2017. Metric based detection of refused bequest code smell. In *9th International Conference on Computational Intelligence and Communication Networks (CICN)*. 53–57.

[16] Matthew James Munro. 2005. Product metrics for automatic identification of" bad smell" design problems in java source-code. In *Software Metrics, 2005. 11th IEEE International Symposium*. IEEE, 15–15.

[17] M J Munro. 2005. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *11th International Software Metrics Symposium (METRICS)*. 15–15.

[18] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621.

[19] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 7.

[20] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *30th international conference on Software maintenance and evolution (ICSME)*. 101–110.

[21] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *28th International Conference on Automated Software Engineering*. 268–278.

[22] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[23] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2015. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering* 41, 5 (May 2015), 462–489.

[24] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. 2016. A textual-based technique for Smell Detection. In *24th International Conference on Program Comprehension (ICPC)*. 1–10.

[25] R Peters and A Zaidman. 2012. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *16th European Conference on Software Maintenance and Reengineering (CSMR)*. 411–416.

[26] K Siva Prasad Reddy. 2017. Spring Boot with Groovy, Scala, and Kotlin. In *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Apress, Berkeley, 259–278.

[27] Ghulam Rasool and Zeeshan Arshad. 2015. A review of code smell mining techniques: Code Smell Mining Techniques. *Journal of Software: Evolution and Process* 27, 11 (2015), 867–895.

[28] PATRIK SCHWERMER. 2018. Performance Evaluation of Kotlin and Java on Android Runtime.

[29] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158–173.

[30] Ashish Sureka. 2016. Parichayana: An Eclipse Plugin for Detecting Exception Handling Anti-Patterns and Code Smells in Java Programs. (Dec. 2016). arXiv:cs.SE/1701.00108

[31] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 329–331.

[32] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. 2018. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 4–14.

[33] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[34] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[35] M Tufano, F Palomba, G Bavota, R Oliveto, M D Penta, A De Lucia, and D Poshyvanyk. 2017. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[36] E van Emden and L Moonen. 2002. Java quality assurance by detecting code smells. In *9th Working Conference on Reverse Engineering (WCRE)*. 97–106.

[37] Bartosz Walter and Tarek Alkhaeir. 2016. The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology* 74 (2016), 127–142.

[38] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer. 236 pages.

[39] Aiko Yamashita. 2014. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. *Empirical Software Engineering* 19, 4 (2014), 1111–1143.

[40] Aiko Yamashita and Steve Counsell. 2013. Code smells as system-level indicators of maintainability: An empirical study. *Journal of System and Software* 86, 10 (2013), 2639–2653.

[41] Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *35th International Conference on Software Engineering (ICSE)*. 682–691.

[42] Aiko Yamashita and Leon Moonen. 2013. To what extent can maintenance problems be predicted by code smell detection? – An empirical study. *Information and Software Technology* 55, 12 (2013), 2223–2242.