

Effectiveness of Kotlin vs. Java in Android App Development Tasks

Original

Effectiveness of Kotlin vs. Java in Android App Development Tasks / Ardito, Luca; Coppola, Riccardo; Malnati, Giovanni; Torchiano, Marco. - In: INFORMATION AND SOFTWARE TECHNOLOGY. - ISSN 0950-5849. - ELETTRONICO. - 127:(2020). [10.1016/j.infsof.2020.106374]

Availability:

This version is available at: 11583/2837799 since: 2020-10-22T16:56:14Z

Publisher:

Elsevier

Published

DOI:10.1016/j.infsof.2020.106374

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

Elsevier postprint/Author's Accepted Manuscript

© 2020. This manuscript version is made available under the CC-BY-NC-ND 4.0 license
<http://creativecommons.org/licenses/by-nc-nd/4.0/>. The final authenticated version is available online at:
<http://dx.doi.org/10.1016/j.infsof.2020.106374>

(Article begins on next page)

Effectiveness of Kotlin vs. Java in Android App Development Tasks

Luca Ardito, Riccardo Coppola, Giovanni Malnati, Marco Torchiano

*Politecnico di Torino
Department of Control and Computer Engineering
Turin, Italy
name.surname@polito.it*

Abstract

Context: Kotlin is a new programming language representing an alternative to Java; they both target the same JVM and can safely coexist in the same application. Kotlin is advertised as capable to solve several known limitations of Java. Recent surveys show that Kotlin achieved a relevant diffusion among Java developers.

Goal: We planned to empirically assess a few typical promises of Kotlin w.r.t. known Java's limitations, in terms of development effectiveness, maintainability, and ease of development.

Method: Our experiment involved 27 teams of 4 people each that completed a set of maintenance tasks (both defect correction and feature addition) on Android apps written in either Java or Kotlin. In addition to the number of fixed defects, effort, and code size, we collected, through a questionnaire, the participants' perceptions about the avoidance of known pitfalls.

Results: We did not observe any significant difference in terms of maintainability between the two languages. We found a significant difference regarding the amount of code written, which constitutes evidence of better conciseness of Kotlin. Concerning ease of development, the frequency of `NullPointerException`s reported by the subjects was significantly lower when developing in Kotlin. On the other hand, no significant difference was found in the occurrence of other common Java pitfalls. Finally, the IDE support was deemed better for Java than Kotlin.

Conclusions: Some of the promises of Kotlin to be a "better Java" have been confirmed by our empirical assessment. Evidence suggests that the

effort in transitioning to Kotlin can provide some advantages to Java developers, especially regarding code conciseness. Our results may serve as the basis for further investigations on the properties of the language.

Keywords: Android, Effectiveness, Kotlin, Java, Maintenance

1. Introduction

Kotlin is a modern programming language, appeared in 2011, which represents an alternative to Java, with which it can seamlessly coexist. Many pieces of evidence are available in the literature underlining that Kotlin is gaining traction among Android software developers. In a previous study, we mined all Android apps hosted on the F-Droid platform and updated after October 2017: we found that nearly one-fifth of them featured Kotlin code, with 2/3 of those projects featuring more Kotlin than Java code [1]. Similar trends have been reported by Oliveira et al. regarding the number of StackOverflow questions about Kotlin programming for Android and GitHub repositories with Kotlin [2].

One of the main design guidelines that led to the development of the Kotlin language is a better handling of null values. In the Java language, without the usage of specific checks, the handling of *null* values can lead to NullPointerExceptions (NPE). Several studies in the literature report the prominent role of NullPointerExceptions among the reasons for Android application to crash. Coelho et al. report that near 30% of all stack traces collected upon the Android app crash contained NPEs as their root causes [3]. The authors also underline the difficulty in protecting the code against those exceptions, especially when the app does not have access to third-party source code. NPEs can also happen – as Payet and Spoto report – in the link between the XML layouts and explicit application code casts [4]. Such a link is obtained utilizing the very commonly used `setContentView` and `findViewById` methods. These method calls are very crucial, and frequent operations are executed every time the components of the application screen are instantiated. The effects of those issues are amplified by misuses of the exception handling mechanisms provided by Java, which are documented frequently among Android developers [5].

Readability and conciseness are considered key-features of the Kotlin language, especially for what concerns the declaration of objects and classes with numerous attributes [6].

32 The novelty of the Kotlin language, and the easiness in adapting existing
33 (and possibly long-running) Java projects to it, suggests the need for an
34 evaluation of the benefits guaranteed to developers from such transition.
35 Many advantages are reported by works in the specialized literature, but to
36 the best of our knowledge, their empirical assessment is still missing. With
37 this work, we aimed at assessing some assumed advantages of Kotlin with
38 respect to Java in the context of Android development and maintenance.
39 To do so, we conducted a controlled study with undergraduate students, a
40 sample that can represent average Kotlin developers due to the low experience
41 possessed – as of today – by developers with such language.

42 In light of an ever increasing adoption of Kotlin for Android development,
43 this empirical assessment aims to provide practical evidence that could help
44 in a transition from Java to Kotlin. In particular we focused on possible ef-
45 fects on maintainability, conciseness, and avoidance of a few common pitfalls.

46 The remainder of the paper is organized as follows: Section 2 provides
47 some background for Kotlin programming, its characteristics, and the recent
48 trends of its diffusion, and it provides a brief review of related work in liter-
49 ature; Section 3 describes the goal, procedure, participants and material of
50 the experiment, along with possible threats to the validity of our findings;
51 Section 4 discusses the threats to the validity of this study; Section 5 reports
52 the results of the experiment, that are discussed in section 6; finally, Section
53 7 concludes the paper.

54 **2. Background**

55 Kotlin first appeared in 2011, but its first stable release was distributed
56 in February 2016. In May 2017, Kotlin became a first-class language on
57 Android, and support was provided by the Android Studio DE since release
58 3.0 of October 2017. The popularity of Kotlin increased rapidly since then.
59 The State of Developer Ecosystem in 2018 shows that Kotlin is mainly used
60 for mobile and Server applications working mainly in Oreo and Nougat in
61 Android, and JDK 8 in servers. According to statistics provided by JetBrains,
62 only around 40% of Kotlin developers have adopted the language for more
63 than one year¹.

¹<https://www.jetbrains.com/research/devecosystem-2019/> Last visited January 2020

64 Kotlin is a statically typed programming language that runs on the Java
65 Virtual Machine (JVM) and fully interoperates with Java: it is possible to
66 mix Kotlin and Java code in the same application, to call Kotlin code from
67 Java code and vice versa [7]. The two languages share several common-
68 alities [2], and the official documentation of Kotlin itself reports its main
69 characteristics by means of comparisons with Java.

70 Kotlin takes a pragmatic approach, such as not re-implementing the en-
71 tire Java collections framework making it compatible with the JDK collection
72 interfaces without breaking any existing project implementations. For exam-
73 ple, Kotlin still supports Java 6 bytecode because almost half of the Android
74 devices still run on it. It is possible to start using Kotlin for small parts of a
75 large project, including a few UI components and simple business logic. The
76 possible coexistence between Kotlin and Java can be deemed as one of the
77 main factors that are fueling the transition to Kotlin for Android developers.

78 As a first example of features that are not supported by Java, Kotlin
79 also allows functions in addition to classes to be first level constructs. In
80 Kotlin, everything is an object, even numeric values that in Java are treated
81 as primitive types. Kotlin provides the ability to extend a class with new
82 features without having to inherit from the class or use any design pattern
83 such as Decorator [8] through special declarations called *Extension Functions*
84 and *Extension Properties*.

85 On the other hand, Kotlin does not feature some characteristics of the
86 Java language, like checked exceptions, static members, non-private fields,
87 and the ternary operator.

88 A complete description of the features of Kotlin is out of the purpose of
89 this paper². The primary objective of our work has been instead to verify
90 some of the peculiarities of Kotlin, mostly regarding the avoidance of common
91 Java development pitfalls [9]:

- 92 • *Nullability*: the typical convention used throughout Java APIs is to
93 let a method return a `null` reference to represent the absence of a
94 result. This approach, when not accompanied by appropriate checks,
95 may lead to NPEs, which reportedly is one of the most common causes
96 for crashes in Android apps [3] [10]. Java 8 introduced the `Optional`
97 class to provide API with a clear way to represent "no result" as an

²A large set of open resources about the Kotlin language is available online at <https://kotlinlang.org/docs/reference/>

98 alternative to returning `null`, but not as a general-purpose solution to
 99 the nullability problem [11]. Kotlin provides a way to declare nullable
 100 variables explicitly (`?`) and a safe-call operator (`?.`) that can be used
 101 in conjunction with the *elvis* operator (`?:`) to avoid most NPEs.

102 Figure 1 reports side-by-side examples of equivalent Kotlin and Java
 103 code. We can observe how Kotlin allows declaring a nullable variable
 104 – by default variables are non-nullable – and to use safe call and elvis
 105 operators to achieve safer and more compact code.

<pre>var bob : Person? = null; //... return bob?.department?.name; // safe call</pre>	<pre>Person bob = null; // ... if(bob!=null) if(bob.department!=null) return bob.department.name; return null;</pre>
<pre>var bob : Person? = null; //... return bob?.name:"<?>"; // ?: elvis</pre>	<pre>Person bob = null; // ... return bob!=null?bob.name:"<?>";</pre>
Kotlin	Java

Figure 1: Nullability examples in Kotlin vs. Java

106 • *Mandatory Casts*: Java often requires several explicit casts to let the
 107 compiler cope with type conversions, this makes code longer and hard
 108 to read, in addition, a wrong cast could be accepted by the compiler
 109 and result into a run-time exception; Kotlin introduced *smart casts* and
 110 a *safe* (nullable) cast operator (`as?`).

111 Figure 2 report an example of a safe cast, in Kotlin and Java. Safe casts
 112 are capable of eliminating the possibility of triggering a `ClassCastException`
 113 at run-time. As it is evident from the comparison, the safe cast in Java
 114 requires a more verbose syntax – that we reported with the usage of
 115 the ternary operator – with respect to that needed by Kotlin. Such a
 116 higher verbosity can be deemed as a deterrent for developers to exten-
 117 sively use the practice of safe casting, hence increasing the likelihood
 118 of generating `ClassCast` exceptions.

119 • *Long argument lists*: the invocation of Java methods uses a strict po-
 120 sitional argument mapping. Therefore methods may require passing

<code>val p: Person? = x as? Person</code>	<code>Person p = x instanceof Person?(Person)x:null;</code>
Kotlin	Java

Figure 2: Mandatory casts examples in Kotlin vs. Java

121 many arguments even if they assume default or null values; writing
 122 overloaded methods might help in such cases, but it may require sig-
 123 nificant effort without covering all cases. Kotlin adopts a solution to
 124 this issue by defining default values for arguments and allowing – in
 125 addition to positional arguments – passing arguments by name. Other
 126 recent languages have adopted similar solutions, e.g., default values for
 127 arguments are provided by Python.

- 128 • *Data Classes*: often, a program requires the creation of classes whose
 129 primary purpose is to hold data. The amount of boilerplate code re-
 130 quired by Java to implement these classes can be relevant. The addi-
 131 tional code can often be mechanically derivable from the data: such
 132 automatic derivation is done by libraries that are not part of the stan-
 133 dard Java library, e.g., in project Lombok³. Kotlin introduced the
 134 *Data Classes* that the compiler is able to process to generate all the re-
 135 quired boilerplate code automatically. In our prior investigations about
 136 Kotlin, we found out that the amount of LoCs savings for a data class
 137 with few fields can be of up to 90% w.r.t. the Java equivalent. An
 138 example of Kotlin class and its Java equivalent is reported in Figure 3.

139 The main contribution of our work is a comparison between Java and
 140 Kotlin in the context of Android Mobile Applications, and specifically when
 141 performing maintenance tasks on apps written in either language. We per-
 142 form this comparison with undergraduate students attending the course of
 143 Mobile Application Development, inspired by the work done by Kosar et
 144 al. [12] for setting up the experiment.

145 2.1. Related Work

146 The present manuscript is related to experiments with students in com-
 147 paring different programming languages or practices, and to literature ded-

³<https://projectlombok.org> Last visited March 2019

<pre> data class User(val name: String, val age: Int) </pre>	<pre> class User private String name; private int age; public User(String name, int age){ this.name=name; this.age=age; } public String getName(){ return name; } public String getAge(){ return age; } public String toString(){ return "User(name="+name+",age="+age+"")"; } public boolean equals(Person){...} public int hashCode(){ ... } } </pre>
Kotlin	Java

Figure 3: Data class example in Kotlin vs. Java

148 icated to the comparison between Kotlin and Java. This section reports a
149 summary of relevant work in these fields.

150 2.1.1. Studies on Kotlin

151 Since Kotlin has been released recently, the literature does not include
152 many works related to this topic.

153 Shah et al. [13] analyzed how to perform code obfuscation with Android
154 applications written in Kotlin. Bryksin et al. [14] discussed a methodology
155 for detecting anomalies – i.e., code fragments are written in ways different
156 from the typical ones for a given language – for Kotlin apps. Skripal et al.
157 proposed an Aspect-Oriented extension for Kotlin [15]. Maeda et al. [16]
158 designed a domain-specific language to specify syntax rules based on Kotlin.
159 Belyakova [17] analyzed the dependencies between different language fea-
160 tures. Mateus and Martinez [7] performed an empirical study on the quality
161 of Android apps written in Kotlin, finding that the quality of Android apps
162 is increased by the usage of Kotlin in terms of the presence of code smells.

163 Flauzino et al. [18] performed a comparative study between Java and
164 Kotlin: they compared 50 Java and 50 Kotlin projects to understand if Kotlin
165 contains fewer smells than Java. Code smells are clusters of negative deci-
166 sions made by developers in software design that do not directly affect the
167 execution flow of a program but are potential causes of future problems, in-
168 creasing the complexity, maintainability, and even the cost of software [19].

169 Their findings support the hypothesis that Kotlin presents fewer code smells
170 than Java. With this paper, however, we did not focus on code smells but
171 on maintenance aspects of code development.

172 Banerjee et al. [20] performed comparisons between the usage of Java and
173 Kotlin for developing Android applications. They conclude that the usage of
174 Kotlin makes the development of Android applications easier while reducing
175 the number of errors and bugs in the code. The principal limit of the work
176 by Banerjee et al. lies in the fact that their assumptions are based only on
177 coding tasks executed by the authors (thus, significant researcher biases can
178 be introduced), and no empirical evidence is provided to support them. The
179 results of the present manuscript are in line with those authors' findings but
180 – to the best of our knowledge – we provide the first empirical assessment of
181 the claimed advantages of the Kotlin language when compared to Java.

182 *2.1.2. General Programming Language Comparisons*

183 A large number of works in the literature have performed programming
184 language comparisons. For instance, Nanz et al. performed comparisons
185 of 22 different programming languages of 4 different families. They found
186 that functional and scripting languages are more concise than procedural
187 and object-oriented language, and that compiled and strongly-typed lan-
188 guages are less prone to run-time failures than interpreted or weakly-typed
189 languages [21].

190 Prechelt performed a comparison of seven programming languages, in-
191 cluding C++, Java, and several scripting languages, concluding that modern
192 scripting languages offer reasonable alternatives to C and C++ even for tasks
193 that require substantial amounts of computations [22].

194 Singh performed empirical studies on programming languages used for
195 scientific computing, hinting that in such context, Fortran may be deemed
196 as preferable over Java or C++ [23].

197 Chen et al. studied the effects of the first programming language that
198 is taught to high-level students, finding that no specific programming lan-
199 guage provided better performance in subsequent courses sustained by the
200 students [24].

201 **3. Experimental design**

202 We designed an experiment to be conducted in the context of a Mobile
203 Applications Development course within a Computer Engineering MSc. de-

Table 1: GQM Template for the study

Object of Study : usage of Java and Kotlin programming languages
Purpose : comparing
Focus : effectiveness in avoiding common pitfalls
Context : maintenance and development tasks performed on Android applications by students
Stakeholders : developers, researchers

204 gree, attended by 108 students. During the course, the students usually
205 attend practical labs where they are required to work together in groups to
206 develop code for a course running project. The experiment took place dur-
207 ing two such labs and involved working on both a small application and the
208 course running project.

209 This section follows the reporting guidelines proposed by Jedlitschka et
210 al. [25] and the APA Manual [26] to organize the discussion of the exper-
211 imental design. More specifically, the following subsections provide details
212 about the high-level goal of the experiment, the participants that were in-
213 volved, the overall experimental design, and the individual research questions
214 that we formulated. For each research question, we report the materials, the
215 procedure, and the metrics that were used to answer them.

216 3.1. Experiment Goal

217 We report the design, goal, research questions, and procedure adopted
218 in this study following the Goal Question Metric (GQM) template [27], as
219 summarized in Table 1. The *goal* of the experiment can be expressed as:

220 *Analyze the usage of Java and Kotlin programming languages*
221 *for the purpose of comparing with respect to their effectiveness in*
222 *avoiding common pitfalls from the point of view of developers in*
223 *the context of maintenance and development tasks performed on*
224 *Android applications.*

225 3.2. Participants

226 The experimental units of the experiment were the student groups formed
227 for the Mobile Applications Development course. The groups were formed by

228 picking students ID randomly, in order to avoid any bias in the composition
229 of the groups that could be introduced by allowing the students to compose
230 the groups as they desired. All the groups were formed by four students
231 enrolled in the course and attending the Computer Engineering MSc degree
232 at Politecnico di Torino.

233 The sample of the experiment is clearly a convenience sample that might
234 be representative of small teams of novice developers.

235 Following recommended good practices [28], the subjects were rewarded
236 with points for participating in the experiment. Based on the correctness of
237 their answers, each subject earned up to a 10% bonus on their assignment
238 grade for the course.

239 *3.3. Design and Procedure*

240 The experiment follows a structure that was standard between subjects,
241 with two treatments and two groups. The participating groups worked in
242 two consecutive lab sessions: the first was a warm-up and is not used for the
243 experiment, the latter constituted the actual experiment.

244 The participating groups have been divided into two sets; each set was
245 administered the tasks with different languages (Java vs. Kotlin) in a differ-
246 ent order in the two sessions. After the second lab session, the participating
247 groups were asked to answer a questionnaire about their experience. The
248 questionnaire had the following objectives:

- 249 i. Understand the characteristics of the group;
- 250 ii. Evaluate the level of understanding of the application that they in-
251 spected;
- 252 iii. Assess the issues encountered by the group;

253 The items of the questionnaire were organized into three different sections
254 corresponding to the aims defined above. We report the full questionnaire in
255 Table 2.

256 The tasks performed by the researchers and the students are summarized
257 in the BPMN diagram reported in Figure 4.

258 We aimed at answering three different research questions, based on the
259 outcomes of the experiments and on the answers to the questionnaire. In
260 Table 3, we briefly report the research questions and the materials, tasks, and
261 questionnaire sections that are used to respond to each of them. Detailed
262 descriptions of the procedure to answer each RQ are reported in the following
263 subsections.

Table 2: Questionnaire Structure

Group	N Question	Type	Options
Context	1 What is your group ID?	String	-
	2 How many people are in your group	Numerical	-
	3 How many of you have worked as professional java developers?	Numerical	-
	4 How many of you have worked as professional developers in other languages?	Numerical	-
	5 On average what is your experience in Java programming?	Ordinal	(i) Less than one year (ii) Between one year and three years (iii) More than three years
	6 In this lab what language has been assigned to your group?	Categorical	Java / Kotlin
	7 Have any of you developed programs using Kotlin?	Categorical	Yes / No
	8 What is the Java knowledge of the most experienced member in your group?	Ordinal	(i) Novice: up to 20 classes projects (ii) Intermediate: 20 to 50 classes projects (iii) Advanced: 50+ classes projects
i	1 How easy was to understand the overall structure of the code	Likert	Very Easy - Very difficult
	2 What is the purpose of class RecordingItem?	Categorical	(i) Manage audio registration (ii) Send registration to server (iii) Store registration in database (iv) Wrap registration data (v) Notify the OS when the registration data changes
	3 How many defects did you find in the App?	Numerical	-
	4 How many defects were you able to fix?	Numerical	-
	5 How long did it take to fix the defect(s)? (In minutes)	Numerical	-
	6 Which classes contained defects?	Categorical	(i) Main Activity (ii) FileViewerAdapter (iii) DBHelper (iv) RecordingItem (v) RecordingService
ii	1 Did the IDE (e.g., autocomplete) help in writing code?	Likert	Very Little — Very Much
	2 Did you often experienced <code>NullPointerException</code> ?	Likert	Never — Very Frequently
	3 Did you often encounter problems with methods having long argument lists?	Likert	Never — Very Frequently
	4 How much the effort required to write classes containing mainly data compare to the added value?	Ordinal	Much Higher — Much Lower
	5 How much the effort required to write code to handle class casts compare to the added value?	Ordinal	Much Higher — Much Lower

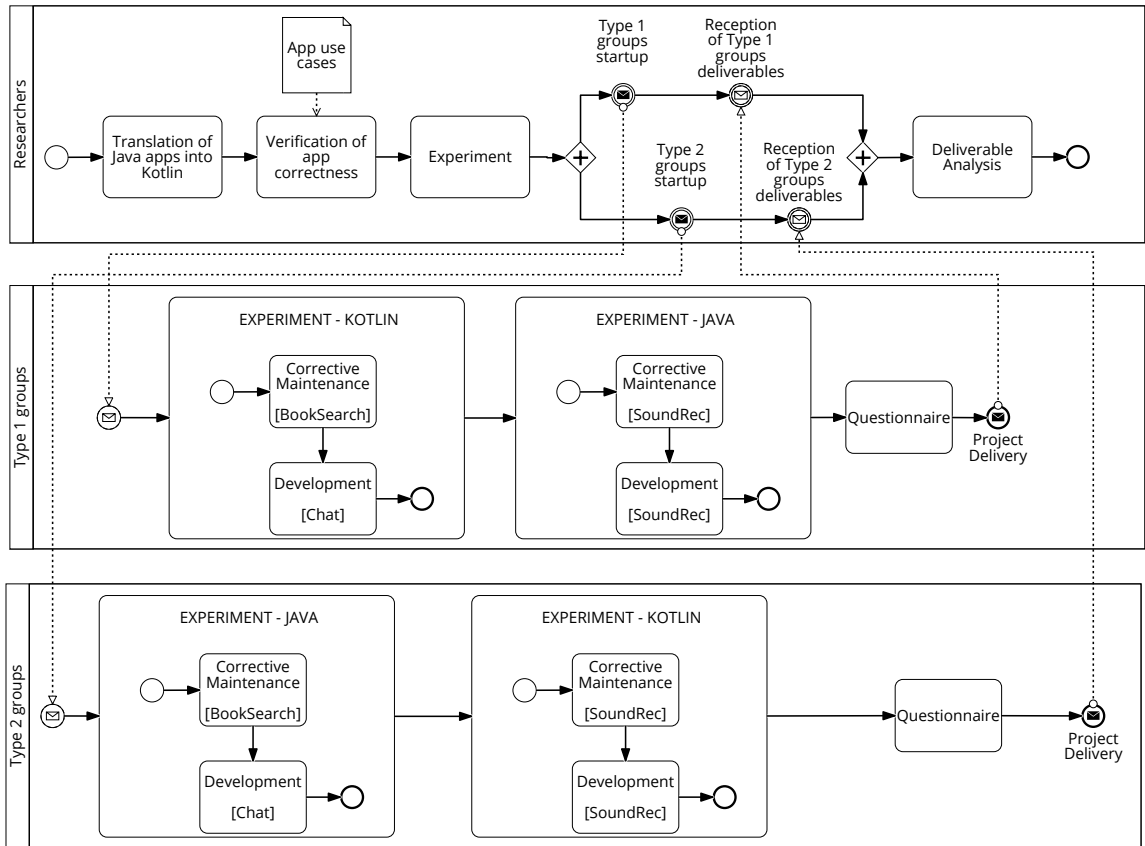


Figure 4: BPMN diagram of the experimental procedure

264 *3.4. RQ1: Maintainability*

265 One of the most cited properties of the Kotlin language is that it makes
 266 code easier to understand and thus more maintainable. We hence formulated
 267 the following Research Question:

268 **RQ1:** *Does the use of Kotlin vs. Java affect the maintainability of Android*
 269 *projects?*

270 In our study, we focus on corrective maintenance, considering the specific
 271 activities of defect location and correction. We aimed, in practice, to under-
 272 stand whether Kotlin’s “better syntax” makes it easier to detect the location
 273 of a defect and the subsequent code change less difficult.

Table 3: Summary of the research questions

RQ	Materials	Experimental Tasks	Analyzed Variables
RQ1: Maintainability	BookSearch SoundRecorder Questionnaire (sec. 1)	Corrective maintenance	Understanding level Defect location accuracy Fix effort
RQ2: Conciseness	Course-running project Questionnaire (sec. 2)	New feature development	Lines of Code No. of Classes
RQ3: Coding Pitfalls	Questionnaire (sec. 3)	Post-experiment reflection	Experienced pitfalls

Table 4: Characteristics of the applications.

App	Java		Kotlin	
	Classes	LOCs	Classes	LOCs
Booksearch	5	471	5	450
SoundRecorder	13	1525	12	1340

274 3.4.1. Materials

275 The maintenance task of the experiment was based on two small appli-
 276 cations (both available in Java and in Kotlin). The main characteristics of
 277 the two apps are shown in Table 4.

278 The first application, Booksearch⁴, uses the OpenLibrary API⁵ to search
 279 book-related information and display their cover images. It is quite a small
 280 application counting just five classes with 471 LoCs. This application was
 281 used in the pre-experiment.

282 The second application, SoundRecorder⁶, is a tool for performing record
 283 and playback of sounds. The application package, albeit larger than Book-
 284 search, can be considered still rather small: it counts 1525 LoCs in 13 classes.
 285 The second task with this application had the purpose of making the student
 286 familiarize with the assigned task.

287 Both applications are open-source and available on GitHub. Since the ap-
 288 plication packages are written using Java, the translation to a Kotlin version
 289 of the application (needed to allow the participants to perform maintenance
 290 tasks in such language) was performed by one of the authors of the paper.

⁴Available at: <https://github.com/shrikant0013/android-booksearch>

⁵Available at: <https://openlibrary.org/dev/docs/api/search>

⁶Available at <https://github.com/dkim0419/SoundRecorder>

291 While the IntelliJ Idea IDE is capable of automatically translating from Java
292 to Kotlin, we opted for a manual translation performed by one of the authors
293 – having more than ten years of experience in Android development – in order
294 to have the code as close as possible to an application natively developed in
295 Kotlin. For each app, we identified the main use cases that were reproduced
296 on the Kotlin versions by another author of the app. The execution of all the
297 use cases allowed to validate the correctness of the translated apps, and to
298 verify that they provide the same functionalities of their Java counterparts⁷.

299 The first section of the questionnaire administered to the participating
300 groups concerned the maintainability concepts measured to answer RQ1.

301 3.4.2. *Experimental tasks*

302 The participants in the experiment had to perform a task of corrective
303 maintenance on the two apps described above.

304 The two applications have been injected with two defects each. The
305 defects are equivalent in both applications, one resulting in a `NullPointerException`
306 exception, and one to an exception caused by a missing element in the layout
307 hierarchy of the application. Specifically, the `NullPointerException`s were
308 obtained by removing instructions to find a specific view (hence resulting
309 in a null view on which the user can operate) and by removing the call to
310 the initialization of a class. The missing element errors were both obtained
311 by commenting method calls used to populate the GUI of the apps. The
312 bugs were injected in the application code by one author of the paper, and
313 their presence - and their ability to cause crashes - was checked by the other
314 authors independently. Both typologies of bugs are frequently mentioned
315 among the most frequently occurring ones for Android applications [29] [30].
316 Hence they can be deemed representative of defects happening during typical
317 (either industrial or open-source) Android app development projects.

318 The goal for the participants was to detect the faults, locate the defects,
319 and fix them.

320 3.4.3. *Hypotheses and Variables*

321 The following null hypotheses were defined to answer RQ1:

⁷We report as a digital appendix the use case narratives of the applications:
https://figshare.com/articles/Effectiveness_of_Kotlin_vs_Java_in_Android_App_Development_Tasks_-_Use_Cases_of_experimental_subjects/11808018

322 Hu_0 There is no difference between the understanding level achieved when
323 using Java or Kotlin.

324 Hl_0 There is no difference between the capability of locating a defect when
325 using Java or Kotlin.

326 Ht_0 There is no difference between the reported time required to correct a
327 defect when using Java or Kotlin.

328 The variables considered in our analysis correspond to the answers col-
329 lected through the questions in group 1 of the questionnaire.

330 Besides, we defined three derived measures, that were automatically com-
331 puted based on the answers to the questionnaire:

332 **Purpose understanding** is defined starting from item *ii.2*. The item asks
333 for a specific class in the application. One of the five options is cor-
334 rect; the others are wrong. *Purpose understanding* is a dichotomous
335 variable whose levels can be either *correct* or *wrong*. More specifically,
336 the RecordingItem class is used in the SoundRecorder app to manage
337 data about recordings; hence the Purpose understanding measure was
338 *correct* for all the experimental subjects that selected the fourth answer
339 to question *ii.2*.

340 **Location accuracy** is defined starting from item *ii.6*. The item asks the
341 respondents to identify the classes where the defects are located.

342 Two out of five classes are expected as a correct answer. We adopt
343 an information retrieval approach and compute the accuracy of the
344 answer.

In particular, *Defect Location Accuracy (LA)* is a ratio measure defined
as:

$$LA = \frac{TP + TN}{TP + TN + FP + FN}$$

345 Where TP are the true positive, TN are the true negatives, FP are
346 the false positives, and FN are the false negatives.

347 More specifically, the two defects of the SoundRecorder app were in-
348 jected in the RecordingItem and FileViewerAdapter classes. Hence,
349 the maximum score for the Defect Location Accuracy was obtained if
350 and only if the respondents checked these two classes only in question
351 *ii.6*.

352 **Fix effort** is defined starting from item *ii.5*, that in the questionnaire col-
353 lects the time employed by each group to fix the defects, and item *ii.6*,
354 that reports the defects supposedly identified by the groups.

355 *Fix Effort* is defined as the ratio of the number of answers checked for
356 question *ii.6* and the time estimated by the group for fixing the defects;
357 hence, it serves as a self-estimate of the average effort (in minutes) to
358 fix one defect.

359 3.4.4. Analysis method

360 Concerning the first research question (RQ1), we analyze three aspects:

- 361 • Understanding: we analyze the *Purpose understanding* variable, and
362 we compare the odds of a correct answer when the program is written
363 in Java vs. Kotlin. To this end, in order to assess Hu_0 , we apply a
364 Fisher test for 2×2 contingency tables that test the null hypothesis
365 that the odds ratio is equal to one.
- 366 • Defect location: we analyze the variable *Location accuracy* to assess
367 Hl_0 , in particular, we apply a Mann-Whitney test to check the null
368 hypothesis that there is no difference between the medians.
- 369 • Time to fix a defect: we analyze the variable *Fix effort* to assess Ht_0
370 by using Mann-Whitney test.

371 3.5. RQ2: Conciseness

372 Another common claim of Kotlin advocates is that it lends itself to write
373 more compact code. This impacts productivity and indirectly, also maintain-
374 ability.

375 We hence formulate the following Research Question:

376 **RQ2:** *Does the use of Kotlin vs. Java makes the code more concise?*

377 We consider conciseness at the macroscopic level, which means less code,
378 both in terms of the number of classes and LoCs.

379 *3.5.1. Materials*

380 The students worked on a larger running project, which is developed
381 throughout the whole course. The running project consists of an app to help
382 people share books. The app had to allow users to sign up easily and set up
383 a basic profile; then users can make books available for sharing, providing all
384 the relevant pieces of information by accessing some shared database. The
385 users can search for shared books and get in contact, via the app, with the
386 book owner in order to arrange the withdrawal and successive return; as a
387 consequence of each sharing, users' reputation must be updated.

388 The average final size of the projects was around 30 to 40 classes per
389 project, with an average total of 6KLOC, including both Java and Kotlin.

390 The second section of the questionnaire administered to the participating
391 groups concerned the conciseness concepts measured to answer RQ2.

392 *3.6. Experimental Tasks*

393 The students were asked on a weekly basis to perform development tasks
394 on the course-running Android project.

395 For the purpose of the evaluation of Kotlin vs. Java usage for the main-
396 tenance tasks of Android applications, we designed two features to be imple-
397 mented by the students.

398 The specific features were defined by one of the authors of the paper, and
399 were designed to be related to the category of the application under devel-
400 opment, compatible with the subjects' expertise with Android, and feasible
401 in the time frame allocated to the experiment.

402 The features that were defined for the participant were: (1) implement a
403 user chat with notifications, using Firebase (referred as CHAT); (2) imple-
404 ment a way to express user ratings for the exchanged books, using a five-star
405 scheme (referred as RATINGS).

406 *3.6.1. Hypotheses and Variables*

407 The following null hypotheses were defined to answer RQ2:

408 H_{c_0} There is no difference between the measured amount of classes written
409 to implement a new feature when using Java or Kotlin.

410 H_{l_0} There is no difference between the measured lines of code written to
411 implement a feature when using Java or Kotlin.

412 *3.6.2. Analysis method*

413 Regarding code conciseness, we focused on two measures collected through
414 static analysis of the submitted experimental assignments:

- 415 • Classes: number of classes developed for the required feature;
- 416 • Lines of Code: LoCs written to implement the required feature;

417 Both above measures were used to assess H_{c_0} by applying a non-parametric
418 Mann-Whitney test.

419 *3.7. RQ3: Coding Pitfalls*

420 One important principle in the design of Kotlin was to avoid several
421 common pitfalls of the Java programming language.

422 In this work, we decided to investigate four of the common pitfalls, i.e.,
423 *Nullability, Mandatory Casts, Long argument list*, and *Data Classes*, and are
424 described in section 2.

425 To evaluate the occurrence of known issues in Kotlin vs. Java program-
426 ming in the context of Android development, we defined our third and final
427 research question:

428 **RQ3:** *Does the use of Kotlin vs. Java effectively avoids the occurrence of*
429 *common pitfalls?*

430 *3.7.1. Materials*

431 The section (ii) of the questionnaire administered to the participating
432 groups (see table 2) concerned their experience with the occurrence of the
433 investigated common pitfalls. The reported occurrence of the pitfalls was
434 used to answer RQ3.

435 We decided to use the answer to those questions as proxies of the actual
436 occurrence of the pitfalls. This choice is due to the limited observability of
437 the teams while performing their development task. In fact, the participants
438 wrote the code on their own machines; therefore it was not feasible to install
439 a monitoring plug-in as it would be possible had they worked on lab devices.

440 *3.7.2. Hypotheses and variables*

441 The following null hypotheses were formulated to answer RQ3:

442 *Hp1₀* There is no perceived difference in terms of the number of NPEs oc-
443 currences with Java or Kotlin.

444 *Hp2₀* There is no perceived difference in terms of the number of casts with
445 Java or Kotlin.

446 *Hp3₀* There is no perceived difference in terms of issues with long argument
447 lists with Java or Kotlin.

448 *Hp4₀* There is no perceived difference in terms of tool support to Java or
449 Kotlin.

450 *Hp5₀* There is no perceived difference in terms of effort required to write
451 data classes.

452 *3.7.3. Analysis Method*

453 To analyze the perceived pitfalls (RQ3), we resort to the responses to the
454 items *ii.1* to *ii.5* of the questionnaire. For each variable, we compute the
455 effect size using the Cliff Delta statistic, and we used the relative confidence
456 interval for deciding about hypothesis rejection.

457 **4. Threats to validity**

458 *External validity* threats concern the generalization of the results. We
459 have considered two real-world (even if small) open-source applications, thus
460 selecting a realistic context for Kotlin or Java maintenance of Android ap-
461 plications. The course running project also has functional requirements that
462 are common among real-world Android apps, so the used software artifacts
463 can be considered as representative of typical Android apps. Hamedani et
464 al. provided a classification of Android apps under twelve different cate-
465 gories [31]. The considered applications in this experiment can be catego-
466 rized under *Office & Business* and *Music & Video*, both belonging to the top
467 three categories that were identified in the study.

468 It is also possible that the results obtained regarding bug-fixing efforts are
469 not generalizable to other categories of defects that are proper for Android
470 applications. However, classifications of Android defects are provided by Hu

471 et al. [29] [30], and NPEs and layout issues are among the most popular
472 categories of bugs.

473 A final threat to the generalizability of results may be linked to how un-
474 dergraduate students may be considered representative of Android developers
475 in general. The use of students as participants in experiments is, however,
476 widely recognized: Sjoeborg et al. [28] report that 50% of the 2,969 exper-
477 iments in 12 leading software engineering journals and conferences between
478 1993 and 2002 used undergraduate students as participants [28]. Carver et al.
479 define a model for conducting a valid empirical study with students (ESWS).
480 They identify research and pedagogical requirements that need to be man-
481 aged while preparing and executing an experiment in a university course. In
482 short, researchers have to make sure that the study is well-integrated with
483 the course goals and materials, give realistic time estimates for experimen-
484 tal tasks, properly motivate the participants without revealing the goals,
485 measures and analysis prior to the study, allow students to give feedback,
486 convince the participants of the relevance of what they are learning, avoid
487 conflicts with students' other commitments, and give students feedback on
488 the results of the experiment [32]. All these guidelines were followed in the
489 conduction of the work documented in this paper. Besides, Carver et al. [32]
490 also provide a checklist to explain when the various activities should occur
491 (i.e., before starting the study, as soon as the study begins, during the study,
492 or after the study is completed). The requirements and checklist provide a
493 useful guide for judging how well a study is integrated into the university
494 course and for judging the reliability of the results. We used that checklist
495 to verify the research and pedagogical goals in this study.

496 *Construct validity* threats concern the relationship between theory and
497 observation. It is not assured that the *Purpose Understanding*, *Location*
498 *Accuracy*, and *Fix Effort* metrics defined in this paper are the best possible
499 proxies for providing answers to the Research Questions identified for this
500 study. We measure conciseness in terms of code size, though shorter code
501 could – at least in principle – bear a higher cognitive load, thus reversing the
502 benefits stemming from more concise code. Considering the specific features
503 introduced in Kotlin, we do not believe this is the case though there is no
504 empirical evidence supporting such belief. As explained in section 3.7.1,
505 we decided to measure the occurrence of pitfalls by means of proxies. In
506 practice, we inferred the actual occurrence on the basis of the reported pitfall
507 manifestation, recorded through the questionnaire. While this choice was
508 dictated by practical feasibility reasons, we have no reason to believe any

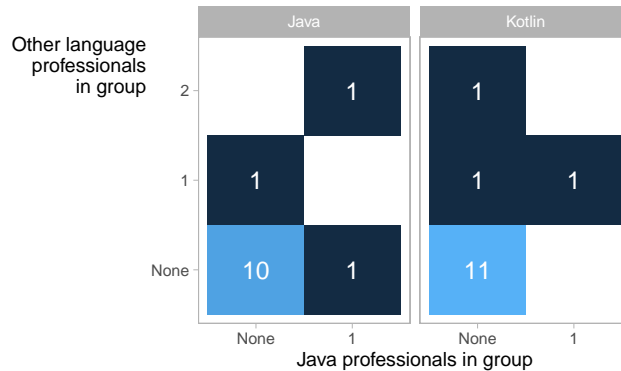


Figure 5: Groups with members having professional experience with Java or other languages

509 misreporting took place. Moreover, we argue the pitfalls do not represent a
 510 problem *per se* but rather in as much they affect the development activities
 511 of the developer, thus in this specific case the reported experience of such
 512 pitfalls is probably closer to the original construct than a mechanical count
 513 of pitfall frequency.

514 As far as the IDE support, we have to notice that in the presence of
 515 participants familiar with Java but not at all with Kotlin, the perceived
 516 support could be more related to familiarity than to actual IDE support.

517 Finally, *Researcher bias* is another possible threat to the validity of this
 518 study, since the authors were involved in the creation of the starting Kotlin
 519 versions of the two considered apps and the bug injection phase. However,
 520 the authors have no reason to favor any language; neither are they inclined
 521 to demonstrate any specific result.

522 5. Results

523 In this section, we report the results measured for the three Research
 524 Questions of the experiment. We also provide details about the population
 525 that participated in the experiment.

526 5.1. Population

527 The population of experimental units that performed all the required ac-
 528 tivities consisted of 27 groups, all made up of four students. Thirteen groups
 529 performed the development tasks in Kotlin, the others in Java. Overall the

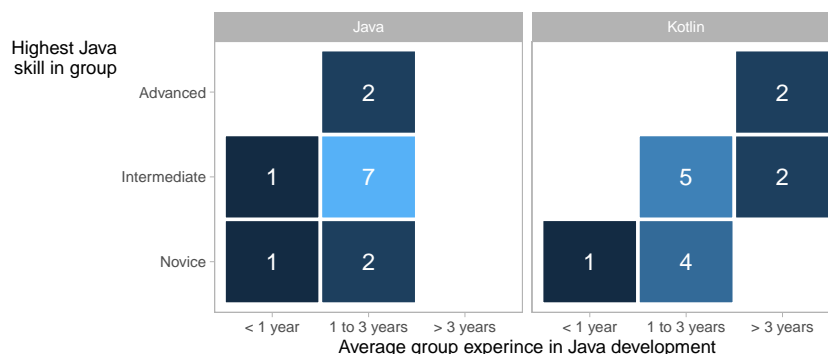


Figure 6: Java skill level and experience of the respondents

530 experiment involved 108 students aged between 25 years old and 40 years old
 531 of different gender and ethnicity.

532 The professional experience of group members was measured through the
 533 answers to question 3 and 4 of the questionnaire. Figure 5 summarizes the
 534 answers to those questions for both typologies of groups. Three groups included
 535 participants that had professional Java development experience, and
 536 overall, 6 out of 27 groups included components with experience as profes-
 537 sional software developers in any language. No participant had any previous
 538 experience in Kotlin.

539 The skill level of the population was measured through the answers to
 540 question 5 and 8 of the Context section of the questionnaire. Figure 6 sum-
 541 marizes the answers to those questions for both typologies of groups. The
 542 experience with Java development was mostly between one and three years,
 543 although three groups had no member with more than one year of experience
 544 in Java, and four groups included a member with more than three years of
 545 experience.

546 In the whole population, four groups had members having advanced Java
 547 skills (i.e., they developed at least one project of over than 50 classes); eight
 548 groups had a most experienced member that considered him/herself a novice
 549 (i.e., they developed a few projects featuring up to 20 classes); in the re-
 550 maining fifteen groups the most experienced member considered him/herself
 551 an intermediate (i.e., at least one medium-sized project of 20 to 50 classes).
 552 Regarding the years of experience with Java, four groups had an average
 553 experience of more than three years, and three groups had an average expe-
 554 rience of less than one year; the remaining groups had an average experience

Table 5: Null hypotheses for RQ1

Name	Description	p-value	Decision
Hu_0	There is no difference between the understanding level achieved when using Java or Kotlin	0.68	Don't Reject
Hl_0	There is no difference between the capability of locating a defect when using Java or Kotlin	0.81	Don't Reject
Ht_0	There is no difference between the reported time required to correct a defect when using Java or Kotlin	0.43	Don't Reject

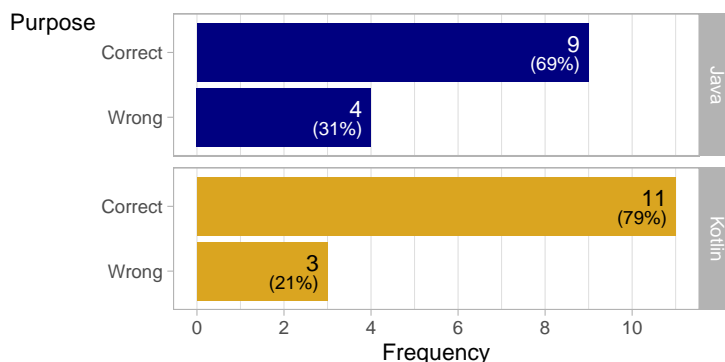


Figure 7: Frequency of correct answer to the purpose understanding question

555 with Java between one and three years.

556 5.2. Maintainability (RQ1)

557 To address the research question concerning maintainability, we focused
 558 on three different aspects: the understanding of the code, the defect detection
 559 ability, and the time required to fix the defects.

560 Table 5 reports the null hypotheses formulated to answer RQ1 and the
 561 related decisions.

562 5.2.1. Understanding

563 Figure 7 reports the measured purposed understanding, based on question
 564 i.2 of the questionnaire. We can observe that four out of thirteen Java groups
 565 failed in the purpose of properly understanding the purpose of a class of the
 566 experimental object. On the other hand, only three out of fourteen Kotlin
 567 groups failed in the same task.

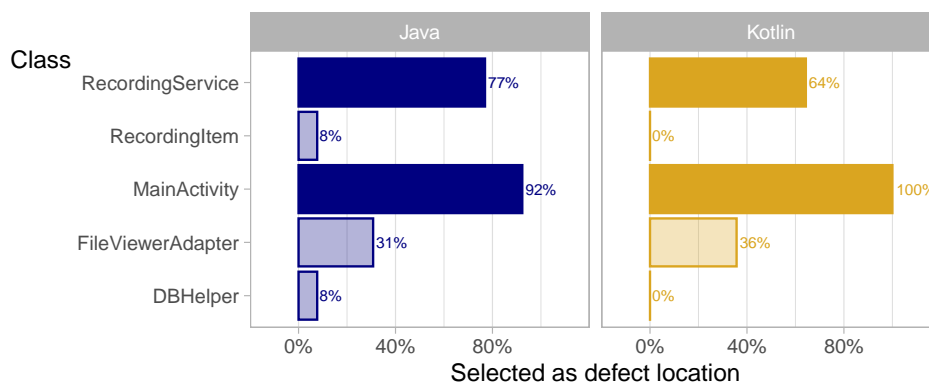


Figure 8: Frequency of the answers to the defect location question selected by the respondents

568 To test the hypothesis H_{u_0} , we performed a Fisher-test that provided a
 569 p-value=0.68. Therefore we cannot reject the null hypothesis.

570 Even though the estimated odds ratio is around 2, such difference is not
 571 statistically significant.

572 5.2.2. Defect location accuracy

573 Concerning the accuracy in defect location tasks, we report in Figure
 574 8 the frequency with which the respondents selected each of the possible
 575 answers to question 6 of the questionnaire. As it can be deduced from the
 576 graph, all groups working with Kotlin and 92% of groups working with Java
 577 were able to find the bug in the MainActivity class of the app. Fewer groups
 578 (respectively 77% and 64% of those working with Java and Kotlin) were able
 579 to spot the other defect injected in the RecordingService class.

580 Figure 9 reports the distributions of the defect location accuracy for the
 581 two languages. We can observe a substantial similarity that is confirmed by
 582 the applied Mann-Whitney test (p-value=0.81). Therefore we cannot reject
 583 H_{l_0} either.

584 5.2.3. Defect correction time

585 To test the hypothesis H_{t_0} concerning the time employed in defect fixing
 586 tasks, we analyzed the average time reported by the groups, normalized by
 587 the number of defects they had found. The distribution of the average time
 588 per found defect is reported in Figure 10.

589 The hypothesis was tested using a Mann-Whitney test that returned a
 590 p-value of 0.43. Therefore we cannot reject the null hypothesis.

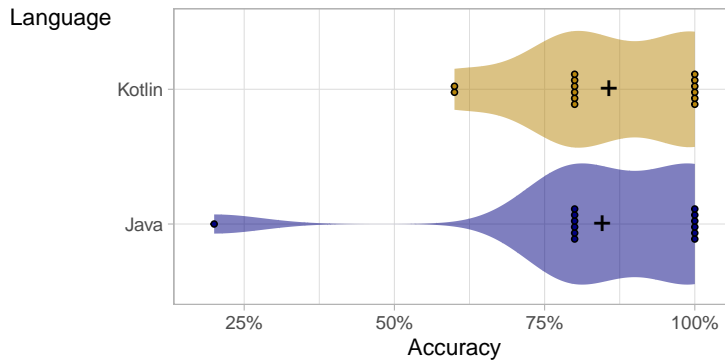


Figure 9: Distributions of the defect location accuracy metric

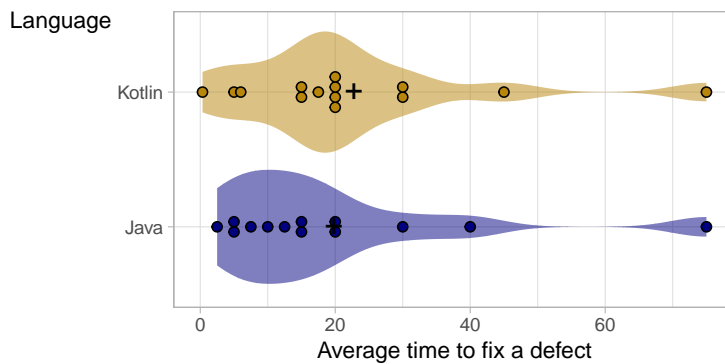


Figure 10: Distributions of the average time to fix a defect

591 The average time to fix defects is similar between Kotlin and Java, with
 592 no statistically significant difference.

593 5.3. Conciseness (RQ2)

594 After the development tasks, we measured both the number of new classes
 595 added to the application design and the lines of new code written overall.

596 Table 6 reports the null hypotheses formulated to answer RQ2, and the
 597 related decisions.

598 In Table 7, we report the measured amount of classes and code that were
 599 added by the respondents to implement the required features. The table
 600 reports the raw count of classes and code and the percentage over the total
 601 amount of code of the application. We also report in the last column the
 602 number of data classes that were developed. The code was automatically

Table 6: Null hypotheses for RQ2

Name	Description	p-value	Decision
H_{c_0}	There is no difference between the measured amount of classes written to implement a new feature when using Java or Kotlin	0.2138	Don't Reject
H_{l_0}	There is no difference between the measured lines of code written to implement a feature when using Java or Kotlin	0.033	Reject

603 measured by a script that leveraged the open-source cloc tool⁸. Blank and
 604 comment lines were not included in the computation.

605 From the table, it can be seen that the number of classes and the amount
 606 of code development had a very high variability between groups. Groups
 607 that worked with Kotlin to implement the new features produced a number
 608 of classes that ranged from 3 to 12 (246 to 1568 lines of code). Four different
 609 groups developed data classes. Groups that worked with Java produced a
 610 number of classes that ranged from 0 to 26 (583 to 4745 lines of code).

611 The distribution of the number of new classes is reported in Figure 11.
 612 We observe a lower number of classes developed for the participating groups
 613 using Kotlin. The mean number of classes developed with Kotlin is 7 while
 614 it is 12 for Java; the medians are respectively 8 and 13.

615 The hypothesis H_{c_0} was tested using a Mann-Whitney test that returned
 616 a p-value=0.2138. Therefore we cannot reject the null hypothesis.

617 The effect size can be considered small, as Cliff's Delta is 0.29; the 95%
 618 CI for the effect size is (-0.24; 0.68): it includes the 0. Therefore, it cannot
 619 be considered as statistically significant.

620 The same kind of analysis can be applied to the amount of Lines-Of-Code
 621 (LOCs) written in order to implement the new feature. The distribution of
 622 the LOCs by language is reported in Figure 12. The median LOCs reported
 623 for Java is between 1526, while for Kotlin, it is Less than 589.5.

624 The hypothesis H_{l_0} was tested using a Mann-Whitney test that returned
 625 a p-value=0.003. Therefore we can reject the null hypothesis.

626 The effect size can be considered large, Cliff's Delta is 0.65, with the
 627 relative 95% CI being (0.24; 0.86); since the CI does not include the 0, the

⁸<https://github.com/AIDanial/cloc>

Table 7: Absolute and relative added classes and LOCs for the development of the required features

Language	Group	Added		
		Classes (%)	LOCs (%)	Data classes
Kotlin	1	7 (17.1%)	483 (8.3%)	0
	3	8 (12.3%)	337 (5.2%)	0
	5	12 (24.0%)	1568 (18.9%)	4
	7	8 (12.7%)	745 (8.1%)	0
	9	5 (4.4%)	515 (4.0%)	0
	11	8 (13.6%)	978 (12.9%)	1
	13	3 (8.8%)	322 (3.2%)	1
	15	9 (20.4%)	664 (10.2%)	0
	17	9 (17.0%)	972 (13.0%)	0
	19	8 (7.1%)	460 (4.0%)	1
	21	5 (9.1%)	380 (5.0%)	0
	23	6 (12.8%)	835 (14.4%)	0
	25	3 (5.1%)	246 (4.3%)	0
	27	9 (14.1%)	744 (7.5%)	0
Java	2	17 (24.3%)	4099 (37.3%)	0
	4	4 (11.8%)	679 (15.6%)	0
	6	17 (37.8%)	1526 (31.6%)	0
	8	22 (31.4%)	2208 (24.9%)	0
	10	26 (32.9%)	4745 (45.0%)	0
	12	17 (30.0%)	2688 (32.0%)	0
	14	0 (0.0%)	583 (7.3%)	0
	16	19 (16.4%)	3810 (31.9%)	0
	18	3 (8.6%)	775 (18.1%)	0
	20	13 (17.6%)	742 (14.9%)	0
	22	6 (15.4%)	1755 (29.6%)	0
	24	9 (15.0%)	1424 (14.7%)	0
26	2 (6.1%)	647 (15.4%)	0	

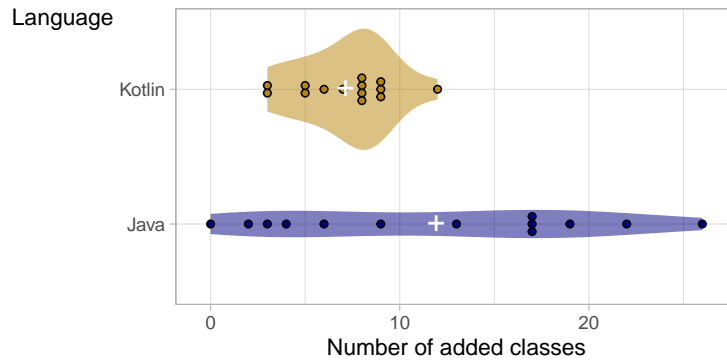


Figure 11: Distribution of number of classes developed.

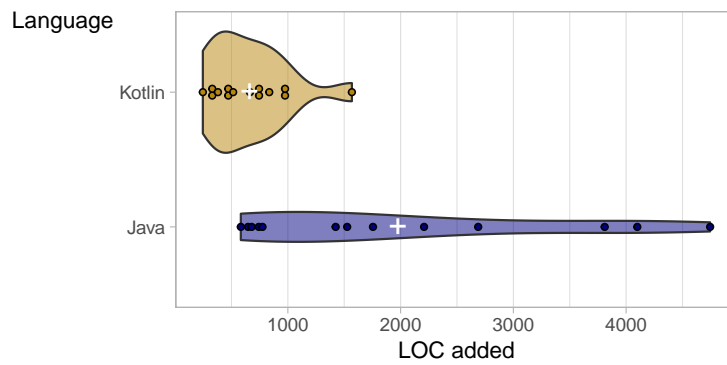


Figure 12: Distribution of LOC written

628 difference can be considered significant.

629 As far as the LOCs are concerned, we can, therefore, reject the null
 630 hypothesis Hl_0 and conclude that there is a significant difference in terms of
 631 the number of LOCs written when developing the same feature with Kotlin
 632 or Java. It is also worth underlining that only four groups used data classes
 633 in Kotlin. Also, one of those groups was the one that wrote most code to
 634 implement the new feature (1568 LOCs). The low number of developed data
 635 classes suggests that the Kotlin language is more concise than Java, even
 636 without taking into consideration the data class construct.

637 5.4. Pitfalls (RQ3)

638 To answer our research questions about the experienced pitfalls by the
 639 developers, we analyzed the self-reported perceptions on the questionnaire.
 640 The null hypotheses for RQ3 are reported in Table 8.

Table 8: Null hypotheses for RQ3

Name	Description	Decision
H_{p1_0}	There is no perceived difference in terms of number of NPEs occurrences with Java or Kotlin	Reject
H_{p2_0}	There is no perceived difference in terms of the number of casts with Java or Kotlin	Don't Reject
H_{p3_0}	There is no perceived difference in terms of issues with long argument lists with Java or Kotlin	Don't Reject
H_{p4_0}	There is no perceived difference in terms of tool support to Java or Kotlin	Don't Reject
H_{p5_0}	There is no perceived difference in terms of effort required to write data classes	Don't Reject

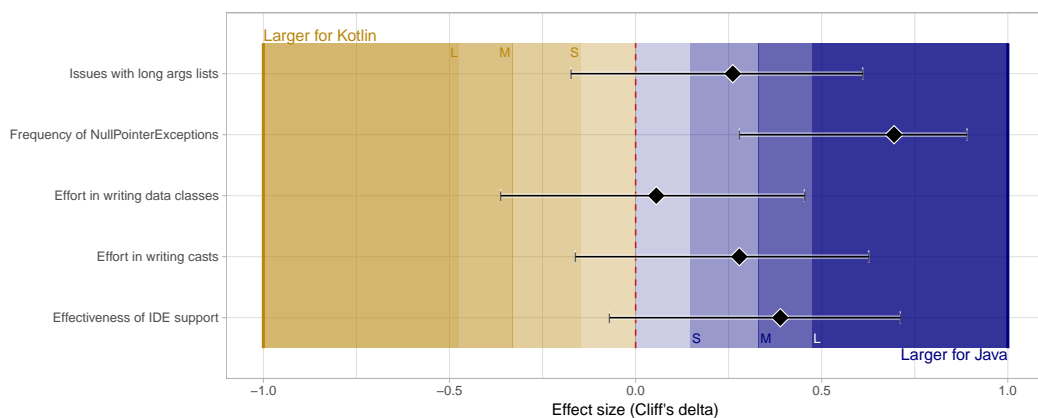


Figure 13: Effect size with confidence interval of language for different aspects.

641 The distributions of the answers to the perception questions are reported
 642 in Figure 13. For each aspect, the figure reports Cliff's Delta estimate effect
 643 size as a diamond shape, and the relative 95% CI is as a whiskered segment.
 644 The shades of the background represent the standard quantification ranges
 645 for the practical magnitude of the effect size.

646 With reference to the 95% confidence interval, the following assumptions
 647 can be based on the respondents' perceptions:

- 648 • More NPEs and null-pointers related issues are likely to happen when
 649 coding with Java; the difference in the respondents' answers is statisti-
 650 cally significant, and the size of the effect is of large magnitude;

- 651 • The usage of Kotlin for writing code casts is perceived as less effort con-
652 suming than Java; this difference is not statistically significant though
653 the effect size magnitude is medium;
- 654 • The respondents considered the definition of long argument lists with
655 Kotlin easier than with Java; this difference is not statistically signifi-
656 cant though the effect size magnitude is medium;
- 657 • Java is perceived as better supported by tooling than Kotlin; this dif-
658 ference is not statistically significant though the effect size magnitude
659 is medium;
- 660 • Negligible differences are observed in terms of the effort required to
661 write data-intensive classes by the developers, in Java or Kotlin.

662 6. Discussion

663 The overall goal of our comparative investigation on Java vs. Kotlin
664 was focused on assessing the consequences of a possible transition from one
665 programming language to the other. The switch could occur at different
666 levels: from a single project to a unit, up to a whole company.

667 We know that, by design, Kotlin is fully compatible at the bytecode-level
668 with Java; therefore, a smooth, progressive transition between the two lan-
669 guages is technically possible. The focus of our research questions addressed
670 the development part of the transition that entails:

- 671 • Understandability of the code written in the new language;
- 672 • Defect location effectiveness;
- 673 • Defect correction efficiency;
- 674 • Code conciseness;
- 675 • Error proneness.

676 6.1. Maintenance (RQ1)

677 The first three former items can be comprised under the broader area of
678 maintenance and were addressed by our first research question (**RQ1**). The
679 results from our experiment show that no specific difference was detected in

680 terms of the ability to detect defects, to fix them, and the effort required to
681 perform the change.

682 *There is no evidence that the use of Kotlin, as a substitute for Java,*
683 *either enhances or lessens software maintainability (RQ1).*

684 It is important to assess how this finding can be generalized. For this
685 purpose, we have to consider the background of the participants in our ex-
686 periment: they are students in a computer engineering master’s degree, only
687 three teams in each language group reported some professional experience.
688 Overall we may consider them close to a junior developer profile. Moreover,
689 we wish to stress that none of our participants had any previous experience in
690 Kotlin. Nonetheless, they were able to detect and fix the defects seamlessly.
691 This may represent evidence in favor of limited risks deriving from the switch
692 to Kotlin in a company, even with little previous knowledge of Kotlin.

693 From our understanding, it remains an open question whether the lack
694 of evidence in terms of maintainability was due to the confounding factors
695 represented by the characteristics of the participants and the small size of
696 the applications.

697 6.2. Conciseness (RQ2)

698 One of the design goals of Kotlin, likewise other recent generation lan-
699 guages, is an increased expressive power that enables writing code in a more
700 concise way. This feature is extremely important because it can improve both
701 the productivity – by reducing the sheer number of keystrokes required – and
702 the understandability of the code. The second research question (RQ2) in
703 our study addressed this aspect. In this respect, we observed a significant
704 effect of using Kotlin on the amount of code written to develop new features.
705 More specifically, we found a large and statistically significant difference in
706 terms of lines of code developed (see Figure 12): Java development required
707 writing three times more lines than Kotlin. Concerning the number of classes
708 created in the development of new features, while the average is 67% higher
709 for Java, the difference is not statistically significant.

710 *We found evidence that the usage of Kotlin led to writing more concise*
711 *code than Java (RQ2).*

712 While we believe that, in general, the adoption of Kotlin can lead to
713 a more concise code, the extent of code reduction depends on the specific
714 type of application and environment. As we mentioned in section 4, our

715 experiment was able to provide evidence limited to the Android environment
716 and specifically concerning two small applications.

717 An important aspect that deserves further investigation is the capability
718 of modern construct that is present in Kotlin – but we argue in many mod-
719 ern programming languages – to actually enable more concise code in many
720 different settings.

721 Also related to the previous research question, we wonder if conciseness
722 stemming from more expressive constructs also translates into a higher un-
723 derstandability of the code.

724 6.3. Error reduction (RQ3)

725 An important selling point of Kotlin, as opposed to Java, is the capability
726 to reduce programming errors through a set of new syntax constructs. Kotlin
727 offers several new constructs, though, in our experiment, we focused our
728 attention on four of them: Nullability, Mandatory Casts, Long argument
729 lists, and Data Classes (see details in Section 2).

730 Our study provides evidence suggesting a reduction in the occurrence of
731 `NullPointerException` during development. For teams using Java, *NPE*
732 occasionally occurred too frequently, while for teams using Kotlin, they hap-
733 pened mostly rarely. Although not statistically significant, we also observed
734 a lesser reduction of issues with long arguments lists and with the effort in
735 writing casts. Finally, no evidence was found of any reduction in the effort
736 devoted to writing data classes.

737 Our respondents also reported slightly better language support for Java
738 than for Kotlin. Even though the difference is not statistically significant, it
739 is surprising considering that the producer of the IDE is the same company
740 that designed the Kotlin language. Probably the much longer experience
741 available for Java development allowed for better support.

742 *We found evidence that Kotlin was able to reduce the frequency of Null-*
743 *PointerExceptions. While no evidence was found concerning effects on other*
744 *investigated pitfalls. (RQ3).*

745 The extent to which these findings can be generalized to experienced
746 programmers is unknown. We can speculate that NPE would represent a
747 lesser problem for experienced developers, though both writings cast and
748 dealing with long arguments lists can be expected to be issued less dependent
749 on the developers' experience. The essentially inconclusive result concerning
750 data classes might be due to the architecture of the application and the

751 characteristic of the required new features, which did not require the use of
752 any data class (see Table 7).

753 Concerning the IDE support, we have to keep in mind that no student had
754 any experience with Kotlin’s development before the experiment. This bias
755 could have affected the participants’ perception. Therefore we could have
756 measured the familiarity with the language rather than the actual support
757 provided by the IDE.

758 The limited evidence and partially counter-intuitive results concerning
759 the coding pitfalls deserve further investigation. Research should be aimed
760 at understanding whether and under which circumstances the adoption of
761 Kotlin allows avoiding the pitfalls.

762 *6.4. Practical implications*

763 The above findings, though preliminary and subject to some generaliz-
764 ability limitations, can bear significant implications. We can summarize such
765 implications for three categories of stakeholders:

- 766 • **Developers** willing the transition from Java to Kotlin: we provide
767 evidence that no negative effect on maintainability can be expected, a
768 more concise code is likely to be written, and that the Kotlin language is
769 able to reduce the occurrence of one of the four pitfalls we investigated.
- 770 • **Researchers** interested in Kotlin: we highlighted a few interesting
771 aspects, regarding some of them we could not get any conclusive re-
772 sult, e.g., the effect on writing casts, long arguments lists, and data
773 classes. Such aspects are good candidates for further studies, as well
774 as confirmative replications of our findings.
- 775 • **Tool builders**: we found some hint of support that is perceived to be
776 better for Java than for Kotlin, further studies could confirm this and
777 provide directions for improving the IDE support.

778 **7. Conclusion**

779 Kotlin is a modern programming language that represents a relevant al-
780 ternative to Java in several development domains. In particular, it has been
781 adopted as an official development language for the Android OS. In this work,
782 we focused on the main promises of this new language. In particular, we in-
783 vestigated how Kotlin can improve the maintainability of code, make code

784 more compact, and avoid common pitfalls. For this purpose, we carried on an
785 experiment in the context of a Mobile Application Development course in an
786 MSc. degree. The experiment compared the Kotlin programming language
787 to its ancestor, Java.

788 With our experiment, we found that the usage of Kotlin apparently does
789 not affect the maintainability with respect to Java, when working on two
790 small applications. At the same time, we found evidence that the adoption
791 of Kotlin leads to more compact code when the subjects of the experiments
792 were asked to develop new features for an ongoing software project.

793 The adoption of Kotlin makes a few common Java annoyances less fre-
794 quent, thus making the development safer. We registered evidence of a re-
795 duction in the frequency of Null Pointer Exceptions. We also observed fewer
796 issues with long argument lists and reduced effort when dealing with casts,
797 although no definitive evidence could be found with this respect.

798 Those findings represent a first empirical assessment of the advantages
799 of Kotlin with respect to Java, as reported by many works in the related
800 literature. The findings showed that most of the promises of the develop-
801 ment of the Kotlin language are reflected by the code produced and by the
802 developers' perception.

803 The study has few limitations, mainly due to the academic settings: the
804 software artifacts were small, the developers were students with limited ex-
805 perience; therefore, the number of bugs and tasks that were studied was
806 limited. The study may not be representative of bigger, real-world projects
807 that require many development tasks and may expose many typologies of
808 defects and issues. It is important to collect more evidence for different and
809 possibly larger applications and outside the Android ecosystem.

810 As future work, we hence plan to investigate the advantages brought
811 by Kotlin in other domains, e.g., server-side development. Also, we aim at
812 finding whether other expected Kotlin benefits hold.

813 **References**

- 814 [1] R. Coppola, L. Ardito, M. Torchiano, Characterizing the transition to
815 kotlin of android apps: a study on f-droid, play store, and github, in:
816 Proceedings of the 3rd ACM SIGSOFT International Workshop on App
817 Market Analytics, pp. 8–14.
- 818 [2] L. M. T. Victor L. de Oliveira, Felipe Ebert, On the adoption of kotlin

- 819 on android development: a triangulation study, in: 27th IEEE Interna-
820 tional Conference on Software Analysis, Evolution, and Reengineering
821 (SANER 2020), IEEE, pp. 1–6.
- 822 [3] R. Coelho, L. Almeida, G. Gousios, A. v. Deursen, C. Treude, Exception
823 handling bug hazards in android, *Empirical Software Engineering* 22
824 (2017) 1264–1304.
- 825 [4] É. Payet, F. Spoto, Static analysis of android programs, *Information*
826 *and Software Technology* 54 (2012) 1192–1201.
- 827 [5] J. Oliveira, D. Borges, T. Silva, N. Cacho, F. Castor, Do android devel-
828 opers neglect error handling? a maintenance-centric study on the rela-
829 tionship between android abstractions and uncaught exceptions, *Journal*
830 *of Systems and Software* 136 (2018) 1–18.
- 831 [6] S. Hellbrück, A Data Mining Approach to Compare Java with Kotlin,
832 *Metropolia Ammattikorkeakoulu*, 2019.
- 833 [7] B. Góis Mateus, M. Martinez, An empirical study on quality of android
834 applications written in kotlin language, *Empirical Software Engineering*
835 24 (2019) 3356–3393.
- 836 [8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Ele-*
837 *ments of Reusable Object-Oriented Software*, Addison-Wesley Profes-
838 sional Computing Series, Pearson Education, 1994.
- 839 [9] VV.AA., *Kotlin Language Documentation*, v 1.3, Technical Report,
840 Kotlin Foundation, 2018.
- 841 [10] A. Levy, Top 5 crashes on android, [https://www.apteligent.com/
842 technical-resource/top-5-crashes-on-android/](https://www.apteligent.com/technical-resource/top-5-crashes-on-android/), 2016. Accessed:
843 2018-02-23.
- 844 [11] B. Goetz, Response to ”should java 8 getters return optional type?”,
845 <https://stackoverflow.com/a/26328555/3687824>, 2014. Accessed:
846 2018-02-23.
- 847 [12] T. Kosar, M. Mernik, J. C. Carver, Program comprehension of domain-
848 specific and general-purpose languages: comparison using a family of
849 experiments, *Empirical Software Engineering* 17 (2012) 276–304.

- 850 [13] Y. Shah, J. Shah, K. Kansara, Code obfuscating a kotlin-based app
851 with proguard, in: 2018 Second International Conference on Advances
852 in Electronics, Computers and Communications (ICAEECC), pp. 1–5.
- 853 [14] T. Bryksin, V. Petukhov, K. Smirenko, N. Povarov, Detecting anomalies
854 in kotlin code, in: Companion Proceedings for the ISSTA/ECOOP 2018
855 Workshops, ACM, pp. 10–12.
- 856 [15] B. Skripal, V. Itsykson, Aspect-oriented extension for the kotlin pro-
857 gramming language, in: CEUR Workshop Proceedings, volume 1864,
858 pp. 1–6.
- 859 [16] K. Maeda, Statically typed domain-specific language to define syntax
860 rules, in: WMSCI 2017 - 21st World Multi-Conference on Systemics,
861 Cybernetics and Informatics, Proceedings, volume 1, pp. 132–135.
- 862 [17] J. Belyakova, Language support for generic programming in object-
863 oriented languages: Peculiarities, drawbacks, ways of improvement, Lec-
864 ture Notes in Computer Science (including subseries Lecture Notes in
865 Artificial Intelligence and Lecture Notes in Bioinformatics) 9889 LNCS
866 (2016) 1–15.
- 867 [18] M. Flauzino, J. Veríssimo, R. Terra, E. Cirilo, V. H. S. Durelli, R. S.
868 Durelli, Are you still smelling it?: A comparative study between java
869 and kotlin language, in: Proceedings of the VII Brazilian Symposium on
870 Software Components, Architectures, and Reuse, SBCARS '18, ACM,
871 New York, NY, USA, 2018, pp. 23–32.
- 872 [19] M. Fowler, Refactoring: Improving the Design of Existing Code,
873 Addison-Wesley, Boston, MA, USA, 1999.
- 874 [20] M. Banerjee, S. Bose, A. Kundu, M. Mukherjee, A comparative study:
875 Java vs kotlin programming in android application development, Inter-
876 national Journal of Advanced Research in Computer Science 9 (2018)
877 41.
- 878 [21] S. Nanz, C. A. Furia, A comparative study of programming languages in
879 rosetta code, in: 2015 IEEE/ACM 37th IEEE International Conference
880 on Software Engineering, volume 1, IEEE, pp. 778–788.

- 881 [22] L. Prechelt, An empirical comparison of seven programming languages,
882 Computer 33 (2000) 23–29.
- 883 [23] D. Singh, An empirical study of programming languages from the point
884 of view of scientific computing, Int. J. Innov. Sci. Eng. Technol 4 (2017)
885 367–371.
- 886 [24] C. Chen, P. Haduong, K. Brennan, G. Sonnert, P. Sadler, The effects of
887 first programming language on college students’ computing attitude and
888 achievement: a comparison of graphical and textual languages, Com-
889 puter Science Education 29 (2019) 23–48.
- 890 [25] A. Jedlitschka, M. Ciolkowski, D. Pfahl, Reporting Experiments in Soft-
891 ware Engineering, Springer London, London, pp. 201–228.
- 892 [26] G. R. VandenBos, Publication manual of the american psychological
893 association (6th ed.), 2010. American Psychological Association, Wash-
894 ington, DC.
- 895 [27] R. Van Solingen, V. Basili, G. Caldiera, H. D. Rombach, Goal question
896 metric (GQM) approach, Encyclopedia of software engineering (2002).
- 897 [28] D. I. K. Sjoberg, J. E. Hannay, O. Hansen, V. By Kampenes, A. Kara-
898 hasanovic, N.-K. Liborg, A. C. Rekdal, A survey of controlled exper-
899 iments in software engineering, IEEE Trans. Softw. Eng. 31 (2005)
900 733–753.
- 901 [29] G. Hu, X. Yuan, Y. Tang, J. Yang, Efficiently, effectively detecting
902 mobile app bugs with appdoctor, in: Proceedings of the Ninth European
903 Conference on Computer Systems, pp. 1–15.
- 904 [30] C. Hu, I. Neamtiu, Automating gui testing for android applications,
905 in: Proceedings of the 6th International Workshop on Automation of
906 Software Test, pp. 77–83.
- 907 [31] M. Reyhani Hamedani, D. Shin, M. Lee, S.-J. Cho, C. Hwang, Andro-
908 class: An effective method to classify android applications by applying
909 deep neural networks to comprehensive features, Wireless Communica-
910 tions and Mobile Computing 2018 (2018).

- 911 [32] J. C. Carver, L. Jaccheri, S. Morasca, F. Shull, A checklist for integrating
912 student empirical studies with research and teaching goals, *Empirical*
913 *Softw. Engg.* 15 (2010) 35–59.

914 **Author Biography**



915

916 Luca Ardito Luca Ardito is an Assistant Professor at Dept. of Control
917 and Computer Engineering at Politecnico di Torino where he works in the
918 Software Engineering research group. He received BSc, MSc, and PhD in
919 Computer Engineering from Politecnico di Torino. His current research in-
920 terests are: mobile development and testing, green software and empirical
921 software engineering methodologies.



922

923 Riccardo Coppola Riccardo Coppola is a Post-Doctoral Research Fellow
924 at Dept. of Control and Computer Engineering at Politecnico di Torino,
925 where he received his MSc and PhD degree in Computer Engineering. He
926 is currently a member of the Software Engineering research group, and his
927 research interests include automated GUI testing for web and mobile appli-
928 cations, and the evaluation of non-functional properties of testware.



929

930 Giovanni Malnati Giovanni Malnati is an Assistant Professor at Politec-
931 nico di Torino. He has participated to several European and national research
932 projects and to many technology transfer activities with private companies,
933 addressing different topics in the areas of embedded, mobile, and multimedia
934 programming. His research activities covers software and network technolo-
935 gies for mobile and pervasive systems, vehicular network applications, indoor
936 positioning systems and multimedia technologies supporting e-learning en-

937 vironments. He is a co-author of seven patents. Since 1999, he cooperates
938 with Istituto Superiore "Mario Boella", participating to a shared laboratory
939 for the development of mobile services and applications. He supervised the
940 research activities of several graduate and PhD students at Politecnico di
941 Torino. He has been the advisor of four PhD students in Computer and
942 Control Engineering and more than 40 master students.

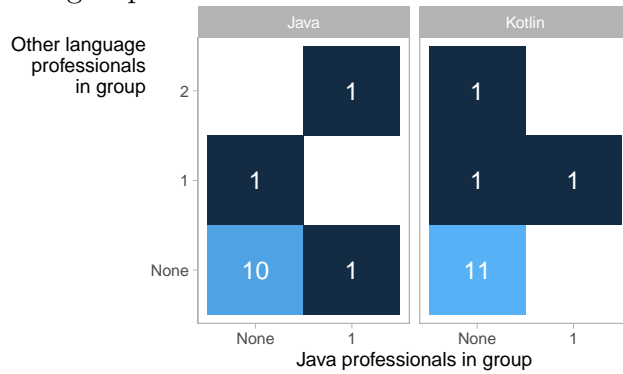


943
944 Marco Torchiano is an associate professor at the Control and Computer
945 Engineering Dept. of Politecnico di Torino, Italy; he has been post-doctoral
946 research fellow at Norwegian University of Science and Technology (NTNU),
947 Norway. He received an MSc and a PhD in Computer Engineering from Po-
948 litecnico di Torino. He is Senior Member of the IEEE and member of the
949 software engineering committee of UNINFO (part of ISO/IEC JTC 1). He
950 is author or co-author of over 140 research papers published in international
951 journals and conferences, of the book "Software Development—Case studies
952 in Java" from Addison-Wesley, and co-editor of the book "Developing Ser-
953 vices for the Wireless Internet" from Springer. He recently was a visiting
954 professor at Polytechnique Montréal studying software energy consumption.
955 His current research interests are: green software, UI testing methods, open-
956 data quality, and software modeling notations. The methodological approach
957 he adopts is that of empirical software engineering.

958 **Appendix**

959 *7.1. Population details*

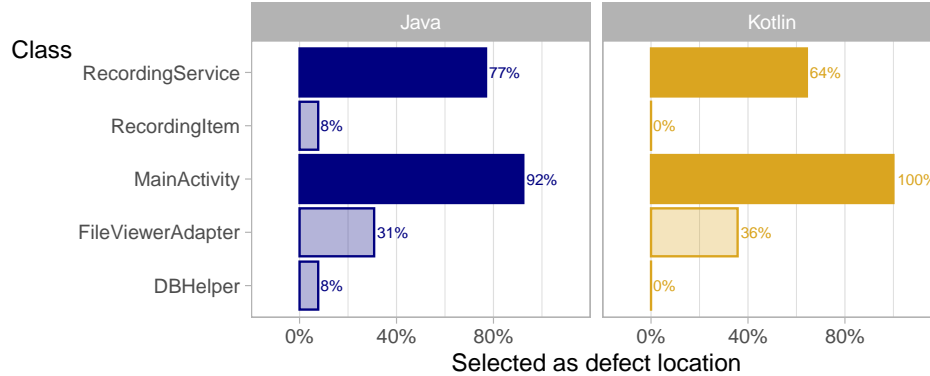
960 Professional experience in Java and other languages in the two experi-
 961 mental groups.



962

963 *7.2. RQ1*

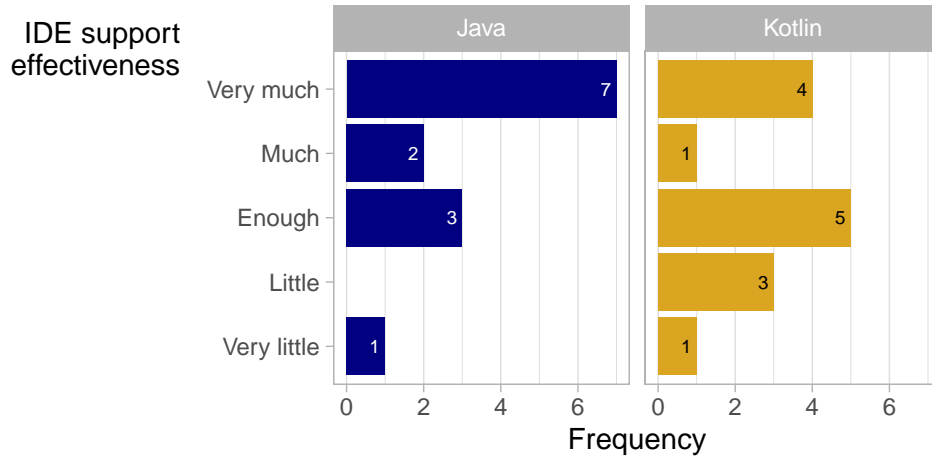
964 Classes selected as location for defects (correct answers, i.e. classes con-
 965 taining actually seeded defects are highlighted)



966

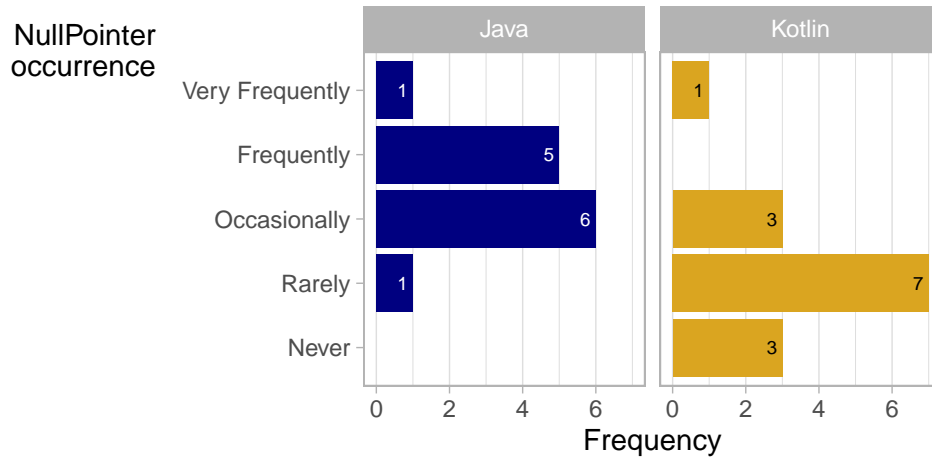
967 7.3. Detailed answer for perceptions

968 7.3.1. IDE support effectiveness



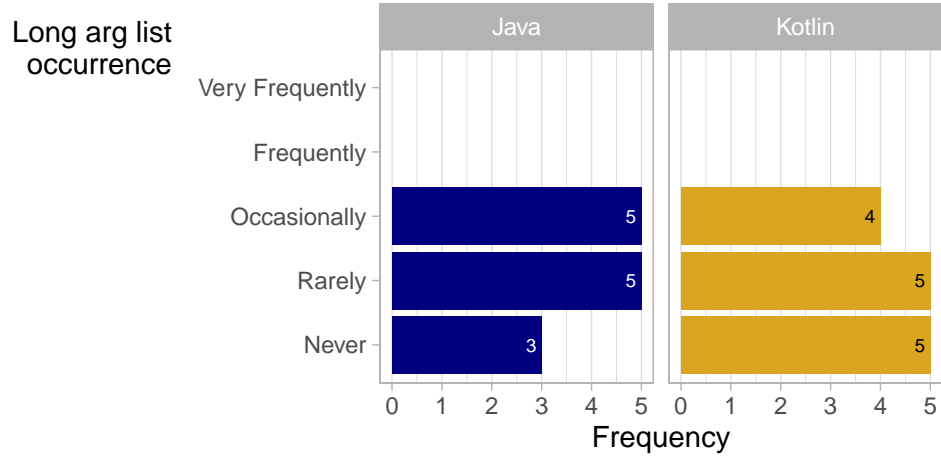
969

970 7.3.2. NullPointerException issues frequency



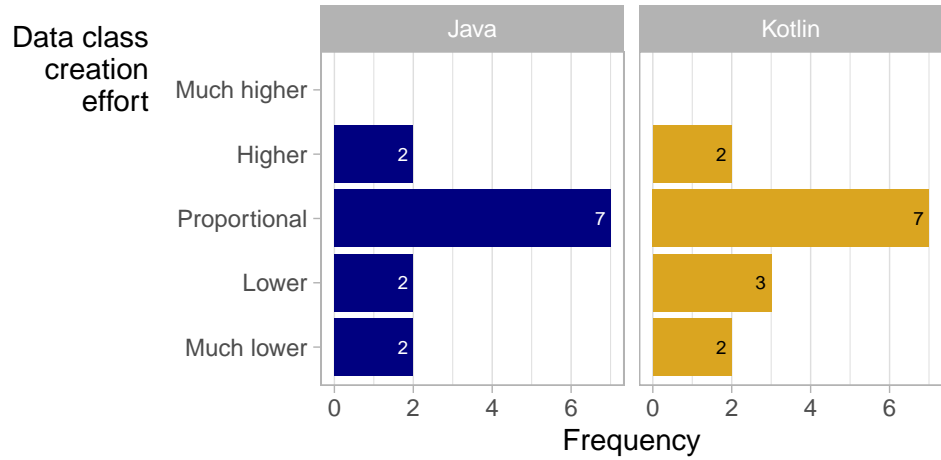
971

972 *7.3.3. Frequency of Long arguments list issues*



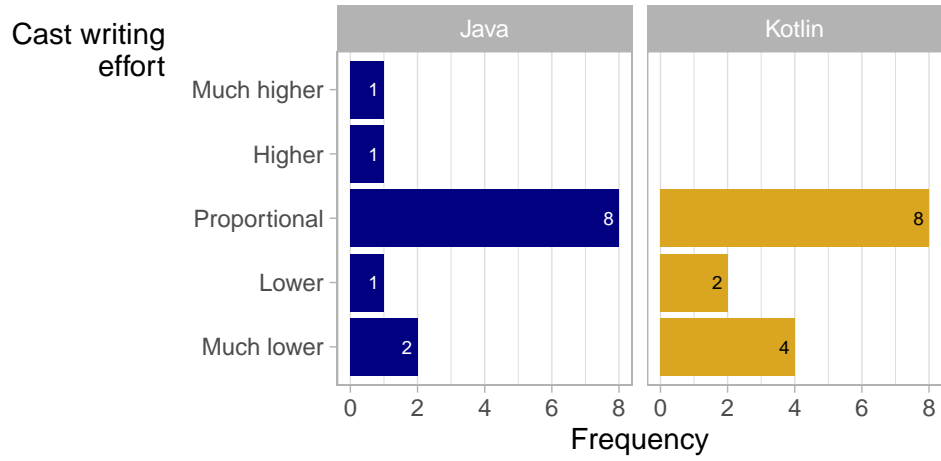
973

974 *7.3.4. Efficacy of data class creation*



975

976 7.3.5. Effort to write casts



977