Contract Usage and Evolution in Android Mobile Applications

David R. Ferreira
Faculty of Engineering, University of Porto, Porto, Portugal

Alexandra Mendes
INESC TEC, Faculty of Engineering, University of Porto, Portugal

João F. Ferreira
INESC-ID & IST, University of Lisbon, Lisbon, Portugal

Carolina Carreira
Carnegie Mellon University, INESC-ID & IST, University of Lisbon, Lisbon, Portugal

Abstract

Contracts and assertions are effective methods to enhance software quality by enforcing preconditions, postconditions, and invariants. Previous research has demonstrated the value of contracts in traditional software development. However, the adoption and impact of contracts in the context of mobile app development, particularly of Android apps, remain unexplored.

To address this, we present the first large-scale empirical study on the use of contracts in Android apps, written in Java or Kotlin. We consider contract elements divided into five categories: conditional runtime exceptions, APIs, annotations, assertions, and other. We analyzed 2,390 Android apps from the F-Droid repository and processed more than 52,977 KLOC to determine 1) how and to what extent contracts are used, 2) which language features are used to denote contracts, 3) how contract usage evolves from the first to the last version, and 4) whether contracts are used safely in the context of program evolution and inheritance. Our findings include: 1) although most apps do not specify contracts, annotation-based approaches are the most popular; 2) apps that use contracts continue to use them in later versions, but the number of methods increases at a higher rate than the number of contracts; and 3) there are potentially unsafe specification changes when apps evolve and in subtyping relationships, which indicates a lack of specification stability. Finally, we present a qualitative study that gathers challenges faced by practitioners when using contracts and that validates our recommendations.

2012 ACM Subject Classification Software and its engineering → System description languages; Software and its engineering → Software development techniques; Software and its engineering → Software verification and validation

Keywords and phrases Contracts, Design by Contract, DbC, Android, Java, Kotlin

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.17

Acknowledgements We thank the anonymous reviewers for their valuable feedback, which helped improve the quality of this paper. This work was supported by Fundação para a Ciência e a Tecnologia (FCT): João F. Ferreira by projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020) and the 'InfraGov' project, with ref. n. 2024.07411.IACDC (DOI: 10.54499/2024.07411.IACDC), funded by the 'Plano de Recuperação e Resiliência (PRR)' under the investment 'RE-C05-i08 - Ciência Mais Digital', measure 'RE-C05-i08.M04' (in accordance with the FCT Notice No. 04/C05 i08/2024), framed within the financing agreement signed between the 'Estrutura de Missão Recuperar Portugal (EMRP)' and the FCT as an intermediary beneficiary; Carolina Carreira by the project VeriFixer, with reference 2023.15557.PEX (DOI: 10.54499/2023.15557.PEX). Alexandra Mendes was financed by National Funds through the Portuguese funding agency, FCT, within project LA/P/0063/2020 (DOI: 10.54499/LA/P/0063/2020).

39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 17; pp. 17:1–17:29
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Building reliable mobile applications is a growing concern, as they are increasingly used in critical domains such as health, finance, and government. There are now more mobile phones than people in the world[1] with more than 2 million apps available in the App Store and Google Play [37]. Additionally, data from 2024 shows that Android is the most used platform (47%), followed by Windows (26%), and then iOS (18%) [35]. Therefore, faults in mobile apps, and particularly in Android apps, can impact a very large number of users. In addition, with an increasing number of apps in critical areas such as health and finance, faults can have a huge negative impact. It is thus important to use software reliability techniques when developing mobile applications.

One of these techniques is Design by Contract (DbC) [26], under which software systems are seen as components that interact amongst themselves based on precisely defined specifications of client-supplier obligations (*contracts*). Suppliers expect that certain conditions are met by the client before using the component (*preconditions*), maintain certain properties from entry to exit (*invariants)*, and guarantee that certain properties are met upon exit (*postconditions*). These contracts are written as *assertions* in the code. Currently, there are assertion capabilities in most programming languages, but assertions are not universally used.

Current efforts in academia and industry show that DbC [27] is an active topic of interest to the software industry, with companies such as Amazon Web Services and Consensys investing largely in the development of tools such as Dafny [25]. Additionally, the creation of tools like Verus [23] for correctness verification in Rust, further underline its importance. Such tools use DbC in the specifications used for formal verification.

DbC can help identify failures [4], improve code understanding [16], and improve testing efforts [36]. This has led to a number of empirical studies on the use of contracts in a variety of contexts [10, 33, 15, 9, 22, 21, 12, 13]. However, *there are no previous studies on the presence and usage of contracts in Android applications nor any study that includes the Kotlin language.*

We present the first large-scale empirical study of contract usage in Android mobile apps written in Java or Kotlin. Our goal is to understand 1) how and to what extent contracts are used, 2) which language features are used to denote contracts, 3) how contract usage evolves from the first to the last version, and 4) whether contracts are used safely in the context of program evolution and inheritance. Information on how practitioners use contracts can help create and improve tools and libraries by researchers and tool builders [33]. Also, empirical evidence about the benefits of contracts can encourage their adoption by practitioners and the establishment of DbC as a software design standard [36].

In summary, the contributions of this paper are:

- The first large-scale empirical study about contract usage and evolution in Android apps, resulting in a list of findings and recommendations for practitioners, researchers, and tool builders. No previous studies consider Kotlin.
- A list of language features, tools, and libraries to represent contracts in Android applications.
- A dataset of 1,767 Java and 623 Kotlin Android apps, together with scripts that can be used to build large-scale datasets of Android apps.

[1] `https://www.weforum.org/agenda/2023/04/charted-there-are-more-phones-than-people-in-the-world/` (last accessed on 01 April 2025)

Table 1 Contract elements considered in this study

category	examples

CREs (74 constructs)	`IllegalArgumentException`
	`EmptyStackException`
	`SecurityException`
	`UnsupportedOperationException`
	`AccessControlException`
	`IndexOutOfBoundsException`
	`NullPointerException`

APIs (31 constructs)	`org.apache.commons.lang.Validate.*`
	`org.apache.commons.lang3.Validate.*`
	`com.google.common.base.Preconditions.*`
	`org.springframework.util.Assert.*`

Assertions (6 constructs)	`assert` (Java) `assert` (Kotlin) `check(), checkNotNull()` (Kotlin) `require(), requireNotNull()` (Kotlin)

Annotations (136 constructs)	`org.jetbrains.annotations.*`
	`org.intellij.lang.annotations.*`
	`edu.umd.cs.findbugs.annotations.*`
	`android.annotation.*`
	`androidx.annotation.*`
	`javax.annotation.*` (JSR305)

Other (1 construct)	`@ExperimentalContracts` (Kotlin)

- An updated and extended version of Dietrich et al.'s tool [13], which can now analyze Kotlin code and can be used to investigate additional Android-specific contracts.
- A user study that validates our recommendations and contributes with further suggestions from practitioners for increasing contract usage.

Even though we update and extend Dietrich et al.'s tool [13], our work *is not* a replication of their study. Our study differs from theirs by focusing on Android apps and not on Java apps only. Due to the focus on Android, our study considers Kotlin in addition to Java, as since 2019, Kotlin is the preferred language for Android app developers[2]. Further, Kotlin is now used by over 60% of Android professional developers[3].

As mentioned above, similar studies to ours have been conducted for different ecosystems, because investigating how developers use contracts can inform future developments that make DbC more effective in practice, thus increasing software reliability.

Data & Artifact Availability. To support our study, an artifact was developed to automatically collect contracts from Android applications and to produce the necessary empirical data. The artifact is written in Python and Java, and includes an extension of the tool proposed by Dietrich et al. [13]. All the code and datasets are publicly available: `https://github.com/sr-lab/contracts-android`

2 Contracts in Android Applications

Our notion of contract follows from the theory of *design by contract* [26], where preconditions, postconditions, and invariants are used to document (and specify) state changes that might

[2] https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred (last accessed on 01 April 2025)
[3] `https://developer.android.com/kotlin` (last accessed on 01 April 2025)

occur in a program. Pre and postconditions are associated with individual methods and constrain their input and output values. On the other hand, invariants are associated with classes and properties and constrain all the public methods in a given class. Preconditions represent the expectations of the contract, and postconditions represent its guarantees. Invariants represent the conditions that the contract maintains.

Contrary to the Eiffel language, conceived by Bertrand Meyer in 1985, neither Java nor Kotlin provide a native and standardized approach for contract specification [10]. Still, developers can take advantage of language features and libraries to specify preconditions, postconditions, and class invariants in both languages. For example, they can use constructs provided by the programming language, such as the Java `assert` keyword introduced in Java 1.4; they can use conditional runtime exceptions such as Java `IllegalArgumentException`; they can use annotations such as the AndroidX annotations `@NonNull` and `@Nullable`; and they can use specialized libraries such as Google Guava's Preconditions API.[4]

To facilitate the comparison with previous studies, we group these constructs into the five categories proposed by Dietrich et al. [13]: conditional runtime exceptions (CREs), APIs, annotations, assertions, and other. The main difference is that, since we focus on Android applications, we include contract elements that are specifically used by Android developers (e.g., Android annotations and specific Android runtime exceptions). To search for relevant contract elements, we used two main additional sources: the Android API Reference[5] and the Kotlin Standard Lib API[6]. Table 1 summarizes the classification and provides some examples; we consider a total of 248 constructs. Below, we briefly describe each category. More details are included in the Supplementary Material [17].

2.1 CREs

An exception can be used to signal, at runtime, a contract violation. Bloch [6] suggests the use of runtime exceptions to indicate programming errors, as the great majority indicates precondition violations. However, it is important to note that the exception itself does not represent a contract; it needs to be associated with a previous check (e.g., an exception thrown inside an *if-else block*) to be considered so. Java and Kotlin offer many exceptions that can be used for this purpose, such as the *IllegalArgumentException*. The *android.util* package offers additional exceptions that we are also interested in analyzing, such as the case of the *AndroidRuntimeException*. Additionally, we are interested in a particular exception, the *UnsupportedOperationException*, which, according to the Java documentation, is thrown to indicate that the requested operation is not supported. As Dietrich et al. argue, this is the strongest possible precondition and can not be satisfied by any client [13].

The following code shows an example of a precondition. An *IllegalArgumentException* is thrown when the contract *shoppingCart.isEmpty()* is violated. The method *proceedWithCheckout* can only perform its task when the *shoppingCart* has at least one item.

```
public void proceedWithCheckout(List<Item> shoppingCart) {
    if (shoppingCart.isEmpty()) {
      throw new IllegalArgumentException();
    }
    ...
}
```

[4] `https://guava.dev/releases/snapshot-jre/api/docs/com/google/common/base/Preconditions.html` (last accessed on 01 April 2025)
[5] `https://developer.android.com/reference` (last accessed on 01 April 2025)
[6] `https://kotlinlang.org/api/core/kotlin-stdlib` (last accessed on 01 April 2025)

We consider a total of 74 CREs (while Dietrich et al. [13] consider six). We show some examples in Table 1 but, due to lack of space, the full list is in the Supplementary Material [17].

2.2 APIs

APIs consist of wrappers around conditional exceptions and other basic constructs. This contributes to a simpler and explicit representation of contracts. We are interested in the four APIs listed in Table 1. For example, the *Apache Commons* offers the *Validate*[7] class that, according to the official documentation, "assists in validating arguments", suggesting a precondition usage. The methods provided by the *Validate* class are simply wrapping exceptions that we have already considered in the CREs. The same libraries do not offer any equal approach to specify postconditions, which suggests a preference from tool builders towards preconditions. Nevertheless, and against the guidelines, practitioners can still use any of those API's methods to check postconditions.

In the following example, that makes use of an API, a precondition *items list is not empty* is declared. In other words, the method *addToShoppingCart* guarantees that if the client fulfills its obligation to provide a non-empty list of items, it will be able to perform its job correctly.

```
import org.apache.commons.lang3.Validate

fun addToShoppingCart(items: List<Item>): List<Item>  {
    Validate.notEmpty(items)
    shoppingCart.addAll(items)
    return shoppingCart
}
```

2.3 Assertions

Assertions were introduced in Java 1.4 and are specified through the *assert* reserved keyword. It helps practitioners verify conditions that must be true during runtime. JVM throws an *AssertionError* if the condition is false. However, JVM disables assertion validation by default, requiring it to be explicitly enabled. This means that practitioners may assume that contracts specified through assertions will be validated at runtime when in fact the assertions are disabled. This leads to an incorrect, and potentially dangerous, assumption. Having that in mind, assertions can still easily be used to check preconditions and postconditions.

In the following example, the contract associated with the *addToShoppingCart* method defines two preconditions — the list of items to add to the shopping cart must have a size of *greater than zero* and *smaller or equal to ten* — and a postcondition — the items added to the shopping cart *will be present in the shopping cart list*.

```
public List<Item> addToShoppingCart(List<Item> items){
  assert !items.isEmpty();
  assert items.size() <= 10;
  shoppingCartItems.addAll(items);
  assert shoppingCartItems.containsAll(items);
  return shoppingCartItems;
}
```

[7] https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html (last accessed on 01 April 2025)

Kotlin also has its own *assert*. However, contrary to the Java version, *assert* in Kotlin is a function and not a reserved word. This means that any class can define a method with the name *assert*, which makes it harder for an automated analysis tool to distinguish between Kotlin's assert or a developer's custom method. Additionally, contrary to Java, Kotlin always executes the assert expression and only uses the *-ea* JVM flag to decide whether to throw the exception. Kotlin also offers other methods: *check()*, *checkNotNull()*, *require()*, and *requireNotNull()*. Although these throw an *IllegalArgumentException* or an *IllegalStateException* instead of an *AssertionError*, we added them to the assertions category because of their syntactic similarities.

The following code uses Kotlin's methods to specify the same pre and postconditions as in the previous Java example.

```
fun addToShoppingCart(items: List<Item>): List<Item> {
    assert(items.isNotEmpty())
    require(items.size <= 10)
    shoppingCartItems.addAll(items)
    check(shoppingCartItems.containsAll(items))
    return shoppingCartItems
}
```

2.4 Annotations

Annotations are metadata added to the program providing information that can be used at compile time or runtime to perform further actions. Java provides many annotations through the *java.lang* package. Table 1 lists the annotation packages we are particularly interested in studying. No previous studies consider the *android.annotation* and the *androidx.annotation*.

The annotation-based approach is particularly interesting for two reasons. First, many annotations can be associated with the method's arguments (preconditions), the method's return values (postconditions), or the class properties (invariants). Second, since annotations are usually added to the method's signature or to the class property, there is a greater separation between the contract specification and the service's implementation. This means that annotations, like in the Eiffel's approach, do not increase the complexity of the method's implementation, contrary to what happens with CREs, APIs, and assertion-based approaches.

The code shown below uses annotations from the *javax.validation.constraints.** packages to specify contracts. The method states that it can only *return a list with a minimum size of 1* (postcondition), if the *item identifier is not null* and the *quantity is greater or equal to one* (preconditions). Also, the class property *items* is associated with a class invariant that states that the *shopping cart can only contain ten items at maximum*. This example shows that adding contracts through annotations does not require adding extra checks to the implementation, contributing to cleaner code.

```
import javax.validation.constraints.*
class ShoppingCart {
    @Size(max=10)
    private val items: List<Item> = mutableListOf()

    @Size(min=1) fun addItem(@NotNull itemUUID: String, @Min(1)
        quantity: Int): List<Item> {
        ...
    }
}
```

2.5 Other

We consider Kotlin Contracts[8], an experimental feature introduced in Kotlin 1.3 that allows developers to state a method's behavior to the compiler explicitly. As the following example shows, they also provide useful information to the compiler: the call to *split* in line 4 causes no error, because the contract specified in line 10 guarantees that *birthdate* is not null.

```
1  @ExperimentalContracts
2  fun sendBirthdayMessage(birthdate: String?) {
3   birthdateIsValid(birthdate)
4   val birthdaySplit = birthdate.split("/")
5    ...
6  }
7
8  @ExperimentalContracts
9  fun birthdateIsValid(birthdate: String?) {
10  contract{returns() implies (birthdate != null)}
11  if (birthdate == null) {
12   throw IllegalArgumentException()
13  } ... }
```

3 Related Work

This section presents related work on the usage of contracts, assertions, and annotations by practitioners.

3.1 DbC and Contract Usage

It is widely supported that DbC contributes to improving software reliability [28, 39, 19]. The advantages commonly mentioned are that DbC (i) improves code understanding [16, 29, 39, 34], (ii) helps identify bugs earlier and diagnose failures [39, 4, 9, 13, 33], and (iii) contributes to better tests [39, 4, 33, 3, 36]. Some studies demonstrated that DbC requires fewer project person-to-hour resources [8, 36], but could not confirm an impact on quality. Moreover, DbC contributes to less time spent on writing tests [36]. Blom et al. [7] suggest that DbC results in fewer errors and decreases development time. In another study, Zhou et al. [42] show that DbC increased reliability in software components. In a study on C# projects using Code Contracts, Schiller et al. [33] found a high percentage of contracts related to null checking and suggest the importance of creating design patterns alongside tools and libraries. Estler et al. [15] analyzed 21 Eiffel, C#, and Java projects known to be equipped with contracts. Most contracts are null checks, with preconditions being typically larger than postconditions. The authors concluded that the average number of clauses per specification is stable over time and that the method's implementation changes more frequently than its specification. However, they warned that strengthening contracts may be more frequent than weakening, indicating some unsafe evolution of contracts. Lastly, Dietrich et al. [13] investigated 176 popular Java projects in the Maven repository and found that the majority of programs do not use contracts significantly. They found that CREs are the most commonly used category, followed by asserts. The dominance of preconditions over postconditions in contracts is consistent with other studies [10, 33]. They found that projects that use contracts maintain or even expand their usage over time. Similarly to Estler et

[8] https://github.com/Kotlin/KEEP/blob/master/proposals/kotlin-contracts.md (last accessed on 01 April 2025)

al. [15], the authors reported some unsafe evolution of contracts, which can happen when a method strengthens its preconditions or weakens its postconditions. They also found many violations of the Liskov Substitution Principle (LSP), with prevalence in the annotations. The LSP states that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. According to this principle, a sub-type can only weaken preconditions or strengthen postconditions and class-invariants from its parent [1]. A sub-type should behave in a way that does not violate the expectations set by its super-type. This ensures that any code that works with the super-type can work with the sub-type without requiring modifications or encountering unexpected behavior. The authors caution that their dataset mainly includes libraries, which may explain the low usage of annotations. This study is the one most related to the work presented here, as it also studies contracts in Java. However, our study differs from Dietrich et al.'s [13] in that, not only we consider more constructs, we also focus on Android apps and we study both Java and Kotlin.

3.2 Assertion Usage

Kudrjavets et al. [22] studied two Microsoft Corporation components, written mainly in C and C++, and found that increased assert density led to a decrease in fault density, and that using asserts was more effective for fault detection than some static analysis tools. Kochhar and Lo [21] studied a dataset of 185 Apache Java projects available on GitHub and found that adding asserts contributes to fewer defects, especially when many developers are involved. This agrees with reports from Kudrjavets et al. [22] but it is not supported by Counsell et al. [12], who analyzed two industrial Java systems and found no evidence that asserts were related to the number of defects. Kochhar and Lo [21] also concluded that developers with more ownership and experience use asserts more often, which shows that more advanced programmers see it as a valuable practice. In line with other previously mentioned studies for contracts [33, 15], most uses are related to null-checking.

3.3 Annotation Usage

There is a general understanding that the use of annotations among practitioners is growing [40, 18]. Yu et al. [40] conducted a study on 1,094 GitHub open-source projects and found a median value of 1.707 annotations per project, with some developers overusing them. The authors argue the need for better training and tools to help derive better annotations. Other authors made a similar claim for contracts [33]. Additionally, developers with higher ownership use annotations more often, which agrees with the findings by Kochhar and Lo [21] related to assertion usage. Grazia and Pradel [18] investigated the evolution of type annotations, some of which can act as contracts, in 9,655 Python projects. The authors reported that although type annotations usage is increasing, less than 10% of potential elements are being annotated. This contradicts the (general) annotations overuse reported by Yu et al. [40]. More importantly, the study found that once added, 90.1% of type annotations are never updated. This indicates that specifications are more stable than implementations, which is desirable. A similar finding was reported by Estler et al. [15] related to the stability of contracts while the program evolves. Also relevant is that most type annotations were associated with parameter and return types, rather than with variable types. Finally, the authors found that adding type annotations increased the number of detected type errors. This motivates the general use of these features to improve software reliability.

4 Study Design

In this section, we present the design of our study, including the research questions, how the dataset of Android apps is created, the classification used for contracts, and the methodologies used to study contract usage and evolution.

4.1 Research Questions

In this study, we aim to answer the following research questions:

- **RQ1**. [**Contract Usage**] How and to what extent are contracts used in Android applications?
- **RQ2**. [**First-To-Last Version Evolution**] How does contract usage evolve in an application from the first to the last version?
- **RQ3**. [**Safety**] Are contracts used safely in the context of program evolution and inheritance?

4.2 Dataset

The dataset used is composed of real-world apps obtained from F-droid,[9] an alternative app store listing over 4,000 free and open-source projects. The fact that it has a large number of open-source apps on a wide range of domains, makes F-Droid a good option. Moreover, F-Droid is normally used in research studies on Android apps [11, 41]. Apart from native Android apps written in Java or Kotlin, F-Droid's catalog also contains projects that use hybrid frameworks (e.g., React Native) that we exclude from our dataset.

We started by downloading the *F-Droid index*, which is a list of URLs for each project available in the catalog. Next, this list is *filtered* based on the following criteria: **1)** The application source code is hosted in GitHub; **2)** The application source code is either Java or Kotlin; **3)** The GitHub project is not archived; **4)** The GitHub project has had a commit since 2018. These inclusion criteria ensure that the project's source code is easily accessible (through GitHub), is written mainly in Java or Kotlin (the languages we are interested in studying), while also guaranteeing that the project is active and relevant. We retrieve *two versions* for each of the filtered projects, which is a required step for the *First-to-Last Version* evolution study. We do this by storing a list of the URLs pointing to two GitHub versions: we first try to retrieve the oldest and the most recent *release*; if there are not enough releases, we try to retrieve the oldest and the most recent *tag*; finally, if there are not enough tags, we just keep the most recent commit of the repository. If there are no releases nor tags, we only consider one version (excluding it from the *First-to-Last Version* evolution study). Although our script resolved most of the versioning schemes found, some projects required manual handling to determine which version was the first and the last. Throughout the paper we refer to the most recent version as *last* or *second* version. Finally, we clone all the projects contained in the versions list. *Every file that is neither a Java nor a Kotlin file is removed* from the dataset, which helps to decrease its size.

4.2.1 Dataset metrics

From the initial list of 4,070 projects in the F-Droid index retrieved on May 21, 2023, we got 3,215 hosted in GitHub, 3,141 non-duplicated URLs, and 2,390 projects after filtering by the

[9] `https://f-droid.org` (last accessed on 01 April 2025)

Table 2 Dataset metrics.

metric	Java	Kotlin	Both
projects	1,767	623	2,390
compilation units	208,479	129,490	337,969
classes	305,749	265,410	571,159
methods (all)	2,113,620	632,416	2,746,036
constructors (all)	208,949	100,534	309,483
methods (public, protected, internal)	1,801,171	506,647	2,307,818
constructors (public, protected, internal)	187,789	99,221	287,010
KLOC including comments	40,635	12,341	52,977

inclusion criteria. Out of these, 1,767 are Java applications and 623 are Kotlin applications. For 1,802 applications we were able to retrieve two versions to be used in the *first-to-last version evolution* study. This means that for 588 applications it was only possible to retrieve one version (these are applications for which there are no GitHub releases nor tags). While these applications are still evaluated in the context of the *usage* and *LSP* studies, they are not considered for the *first-to-last version evolution* study.

Table 2 presents additional metrics about the dataset size. As the table shows, the dataset is imbalanced, with more Java apps. The dataset includes 208,479 Java and 129,490 Kotlin compilation units and, therefore, Java represents 61.7% of the overall number of compilation units. This imbalance requires caution when trying to read this work's results from the perspective of comparing Java against Kotlin's use of contracts. Furthermore, the dataset includes 571,159 classes, 2,746,036 methods, and 309,483 constructors. We did not consider *private* methods, because those are not used directly by a client, and a contract is a bond between a supplier and a client. In total, we analyzed 2,594,828 *public*, *protected*, and *internal* methods and constructors.

In terms of diversity, the dataset includes apps from various domains, such as gaming, communication, multimedia, security, health, and productivity.

4.3 Data Collection and Analysis

Here, we describe the analysis tool and the studies conducted to answer our research questions: the usage study, the *first-to-last version* evolution study, and the Liskov Substitution Principle study.

4.3.1 Analysis Tool

Our analysis tool is an extension of the tool created by Dietrich et al. [13], which was used in their study on the usage of contracts in Java apps. We extended the tool to support Kotlin and more constructs focused on Android apps. Additionally, the framework suffered considerable refactoring and organization to ease its comprehension and maintainability. The main effort was to add support for Kotlin. The original tool used the JavaParser[10] library to perform AST analysis of Java code. Since this library is not able to parse Kotlin source code, we integrated JetBrains's Kotlin compiler[11] to perform this task. This required us to implement new versions of the tool's extractors and visitors classes using the methods

[10] `https://javaparser.org` (last accessed on 01 April 2025)
[11] `https://github.com/JetBrains/kotlin` (last accessed on 01 April 2025)

provided by the new library to be able to identify contract patterns in Kotlin. We also updated the JavaParser library to support newer Java versions.

The tool is divided into three parts: 1) *usage*, which extracts the list of contracts present in each program and produces statistics about their use; 2) *inheritance*, which identifies contracts in overridden methods and validates whether they violate the Liskov Substitution Principle; and 3) *first-to-last version evolution*, which analyses how identified contracts evolve in later versions of the application. The following sections describe how each component contributes to answering our research questions.

4.3.2 Usage Study

The usage study is divided in two main steps: 1) identifying contract occurrences and 2) producing statistics about those results. Our tool uses the JavaParser and JetBrains's Kotlin compiler libraries to perform AST analysis. This analysis is done against a set of extractors to identify occurrences of our defined constructs. Each category requires different approaches for their identification:

- *CREs.* During the AST analysis, we look for the pattern:

  ```
  if (<condition>) { throw new <exception> (<args>) }
  ```

  When this pattern is found, we check whether the exception belongs to the list of CREs considered (see Section 2). In line with Java's good practices, we assume that CREs are used with preconditions.
- *APIs.* Firstly, we check whether the file contains an import declaration to any API package considered. If any is found, all call expressions in that file are analyzed to determine if they are invoking any of the methods provided by the API. As stated before, we assume the analyzed APIs to be associated with preconditions.
- *Assertions.* Identifying Java asserts is straightforward since the JavaParser provides a visitor method for this particular statement. The complexity lies in identifying Kotlin asserts, which is not a reserved keyword. To handle this challenge, when analyzing a file, we first search for any method declaration and any import statement that has a name equal to one of the following expressions: *assert*, *require*, *requireNotNull*, *check*, and *checkNotNull*. Next, we identify whether the class invokes any method with one of those names. Suppose a class contains a method declaration or import statement, as well as an invocation using the name of one of these expressions. In that case, we consider it an ambiguous situation, and therefore, we do not consider it an assert instance. If the class invokes one of those methods but does not declare/import any method with that same name, we consider it an assert. We do not classify assertions either as preconditions or postconditions.
- *Annotations.* We check if the source code file contains an import statement to one of the packages listed in Table 1. If that is the case, we check every annotation in that file to see if it matches any of those provided by the imported package. We also identify the artifact to which the annotation is associated as follows: 1) annotations associated with a method's parameters are preconditions; 2) annotations associated with a method are postconditions; and 3) annotations associated with a field are class invariants.
- *Others.* This category only includes the investigation of the experimental *Kotlin Contracts*. To identify occurrences of this construct, we look for the pattern `contract {returns (<condition>) implies (<condition>)}`.

```
    public static void setToolbarContentColorBasedOnToolbarColor(
        @NonNull Context context,
-       Toolbar toolbar,
+       @NonNull Toolbar toolbar,
        @Nullable Menu menu,
        int toolbarColor,
        final @ColorInt int menuWidgetColor
```

Listing 1 Example of a precondition strengthened using the annotation `@NonNull`, taken from the project Retro Music Player, a music player for Android (in class ToolbarContentTintHelper).

Our tool creates a *JSON* file for program version that stores the identified contracts, including 1) the file path, 2) the associated condition, 3) the method or property name, 4) the type of artifact (method or property), 5) the line number, and 6) the contract type. In the second step of the *usage* study, all the *JSON* files are analyzed to produce statistics about the identified contracts, including the frequency of each category (API, annotation, assertion, etc.), class (preconditions, postconditions, and class invariants), and construct (java assert, Guava API, *androidx* annotations, etc.). For each category, we also compute the Gini coefficient and the list of programs with more contracts.

4.3.3 First-to-Last Version Evolution

We focus on the initial and final GitHub versions of each project as these represent critical moments in the development: the initial introduction of the DbC constructs and the culmination of the development process. This allows us to check if there were any significant changes in the use of contracts.

After identifying a contract in the first version of the app, we check whether, in the later version, the contract still exists, was modified, or removed. We also report cases when a contract is added to an artifact (method or parameter) in the later version of the app (but was not present in the first version). These provide insights into how contracts evolve in an app and whether this evolution poses risks to the client.

As already mentioned, a contract establishes rights and obligations between clients and suppliers. Therefore, when a contract is altered, both parts should be informed and updated accordingly. This is particularly crucial when a *precondition is strengthened* or when a *postcondition is weakened*. In the first case, if the precondition is strengthened and the client does not know it, it can fail to cover its new obligations, and, therefore, the supplier is not bound to keep its part of the contract. In the latter case, if the postcondition is weakened, the client may still be making assumptions that the supplier does not ensure anymore. An example is shown in Listing 1, where the annotation *@NonNull* was added to the *toolbar* parameter in the last version. This is the case of a *precondition strengthening*: in the first version, the method accepted a null *toolbar*, but now it requires it to be not null. Therefore, if the client is not updated, it will fail to cover its new obligation.

Similarly to Dietrich et al. [13], we create *diff records* from the contracts present in the two versions of a program's method and then classify them according to the *evolution patterns* listed in Table 3.

4.3.4 Liskov Substitution Principle Study

When a method is overridden in a subclass, that class can specify new contracts added to the ones inherited from the superclass method. In this case, proper handling of contracts should

Table 3 Classification of the diff records produced during the evolution and LSP study.

Classification	Description	Risk
PreconditionsStrengthened	A precondition was added to a method or a clause to an existing precondition with the '&' or '&&' operators.	Potential risk
PreconditionsWeakened	A precondition was removed from a method, or a clause was added to an existing precondition with the '|' or '||' operators.	No risk.
PostconditionsStrengthened	A postcondition was added to a method or a clause to an existing postcondition with the '&' or '&&' operators.	No risk.
PostconditionsWeakened	A postcondition was removed from a method, or a clause was added to an existing postcondition with the '|' or '||' operators.	Potential risk.
MinorChange	Contract elements are the same, but in different order; or removal of a Nullable postcondition, which is not considered as a significant weakening [13].	No risk.

follow the Liskov Substitution Principle (LSP), which states that the subclass method must accept all input that is valid to the superclass method and meet all guarantees made by the superclass method. In other words, a subclass method can only *weaken preconditions* and *strengthen postconditions*.

To detect those occurrences, we list all methods in each program-version pair associated with their respective class. We also identify the class' parents. Then, similarly to the *first-to-last version* evolution study, diff records are created between the subclass and the superclass methods. These records are classified based on the evolution patterns outlined in Table 3, following the categories and descriptions proposed by Dietrich et al. [13].

5 Results

In this section, we present the results of our empirical study, as well as the main findings. As mentioned earlier, the dataset contains an imbalanced distribution of compilation units, with 61.7% written in Java and 38.3% in Kotlin. This imbalance should be considered when interpreting the findings, particularly in the context of comparing contract usage between Java and Kotlin.

5.1 RQ1: Contract Usage

Table 4 shows the number of contracts found per category, considering all versions (columns 2 and 3) and considering only the latest version of each app (columns 4 and 5). The table also identifies the number of apps containing at least one contract for that category (columns 6 and 7). The most obvious conclusion is that, in both languages, annotation-based contracts are the most popular category. More specifically, considering both languages in the last version, annotations represent 85.2% of the contracts found, followed by CRE with 11.1%, and then assertions with 2.9%. The results show similar tendencies between Java and Kotlin, and the only difference is that while Java's second most popular category is CREs, in Kotlin, it is assertions. This relatively high percentage of the assertion category in Kotlin is explained

Table 4 Number of contracts found in the dataset by category.

	contracts (all ver.)		contracts (2nd ver.)		applications	
Category	Java	Kotlin	Java	Kotlin	Java	Kotlin
API	1,813	10	1,125	9	24	4
annotation	194,448	26,849	115,861	17,490	1,227	547
assertion	3,525	3,868	2,217	2,370	325	234
CRE	26,076	3,374	15,195	2,187	787	288
other	-	1	-	1	-	1

Table 5 Gini coefficient by category.

Category	**Java**	**Kotlin**
assertion	0.70	0.71
API	0.80	0.37
annotation	0.87	0.76
CRE	0.77	0.67
others	-	1.00

by our inclusion of the four language's standard library methods listed in Section 2, where *require()* alone counts 901 total occurrences.

Finding 1: Most contracts are annotation-based, accounting for 86.21% in Java and 79.29% in Kotlin of the total number of contracts found.

This distribution in categories' popularity significantly differs from the findings of Dietrich et al. [13], who reported that the most common category was CREs and found surprisingly low use of annotations. This may be explained by the fact that, while our dataset is formed mostly by user-focused Android applications, Dietrich et al.'s dataset was mainly Java libraries. In Table 6, we can also see that most annotations found belong to the *androidx.annotation.** package that the authors did not consider since it is Android-specific. Nevertheless, the high number of annotation-based contracts found is in line with literature that supports its increasing popularity [40, 18].

From Table 4, we also verify that the usage of *APIs* is low in both languages, and it is even more residual in Kotlin applications, where only nine instances were found in the latest versions. Skepticism around adding third-party dependencies to projects, which may lead to maintainability and support issues in the future, may explain this finding [5, 38].

Finding 2: The use of APIs to specify contracts is rare.

Table 6 shows the frequency of each construct. We highlight that the high number of annotations found is leveraged mostly by the *androidx.annotation.** package. In APIs, the *Guava* library constitutes most of the usage. We were not expecting to see any usage of *Spring Framework Asserts* since this library was designed to be used in the *Spring* framework, but we still found one occurrence. At the same time, we found no occurrences of the now deprecated *FindBugs* annotations. Additionally, we identified a single occurrence of *Kotlin Contracts*, which may depict the practitioner's distrust of using a feature still in an experimental phase.

We now consider Table 5, which presents each category's computed *Gini coefficient*. The *Gini coefficient* measures the inequality among the values of a frequency distribution. In

Table 6 Number of contracts found in the dataset by construct and category.

Construct	Category	contracts (all ver.) Java	contracts (all ver.) Kotlin	contracts (2nd ver.) Java	contracts (2nd ver.) Kotlin
cond. runtime exc.	CRE	25,565	3,232	14,887	2,071
unsupp. op. exc.	CRE	511	142	308	116
java assert	assertion	3,525	-	2,217	-
kotlin assert	assertion	-	3,868	-	2,370
guava precond.	API	1,798	10	1,121	9
commons validate	API	14	0	3	0
spring assert	API	1	0	1	0
JSR303, JSR349	annotation	0	0	0	0
JSR305	annotation	4,195	20	2,133	13
findbugs	annotation	0	0	0	0
jetbrains	annotation	2,310	138	1,596	98
android	annotation	12,003	5,704	7,013	3,414
androidx	annotation	175,940	20,987	105,119	13,965
kotlin contracts	others	-	1	-	1

other words, a *Gini coefficient* of 0 indicates perfect equality, where all apps have the same number of contracts. In contrast, a *Gini coefficient* of 1 means that a single program has all the contracts. We observe that all coefficients in the table are high, except for Kotlin's API usage. This means that although some apps use contracts intensively, the majority does not use them significantly. This aligns with the results found by Dietrich et al. [13]. This conclusion can also be seen in Table 7, where the five projects that use more contracts per category are listed. The table shows the number of contract elements used and the application's category. We find that a small group of projects own a large percentage of the overall use in each category. It is clearly visible from the *assertion* and *CRE* categories that the numbers quickly decrease through the first to the fifth application showing the unbalanced usage between applications. F-Droid does not provide statistics, such as downloads, but the categories shown provide an indication of their purpose (with over half of these applications belonging to the category *Internet*).

Finding 3: Although there are some applications that use contracts intensively, the majority do not use them significantly.

Lastly, Table 8 presents the frequency of each contract type. Once again, we have distinct results for Java and Kotlin. In Java, we found 64.80% of the *classified* instances in the last versions to be preconditions, 22.87% postconditions, and only 12.32% class invariants. These results align with other studies on contracts [10, 33, 13] that show a clear preference towards preconditions. However, results for Kotlin are different: considering last versions, we found 38.81% to be postconditions, 31.64% class invariants, and 29.55% preconditions. This suggests that Kotlin developers tend to favor postconditions, while preconditions come at the last position. According to the classification described in Section 4.3.2, only annotations are classified as postconditions or class invariants. This means that in Kotlin, there is a higher number of annotations associated with methods' return values and class properties than with the methods' parameters.

Table 7 Top five applications using contracts (second versions only) by category.

Category	Applications
assertion	K1rakishou-Kuroba-Experimental (378; Internet), a-pavlov-jed2k (314; Internet), abhijitvalluri-fitnotifications (143; Connectivity), thundernest-k-9 (114; Internet), mozilla-mobile-firefox-android-klar (95; Internet)
CRE	redfish64-TinyTravelTracker (1,036; Navigation), nikita36078-J2ME-Loader (690; Games), abhijitvalluri-fitnotifications (561; Connectivity), lz233-unvcode-android (561; Writing), cmeng-git-atalk-android (447; Internet)
API	wbaumann-SmartReceiptsLibrary (534; Money), alexcustos-linkasanote (318; Internet), BrandroidTools-OpenExplorer (69; System), snikket-im-snikket-android (60; Internet), oshepherd-Impeller (33; Internet)
annotation	MuntashirAkon-AppManager (5,957; System), Forkgram-TelegramAndroid (5,552; Internet), Telegram-FOSS-Team-Telegram-FOSS (5,549; Internet), MarcusWolschon-osmeditor4android (4,393; Navigation), NekoX-Dev-NekoX (4,032; Internet)
other	zhanghai-MaterialFiles (1; System)

Table 8 Number of contracts found in the dataset by type.

	contracts (all ver.)		contracts (2nd ver.)		applications	
Type	Java	Kotlin	Java	Kotlin	Java	Kotlin
precond.	145,961	9,323	85,627	5,810	1,132	355
postcond.	49,694	11,669	30,224	7,632	925	438
invariants	26,623	9,217	16,280	6,221	677	359
unclassified	3,584	3,893	2,267	2,394	279	202

Finding 4: Java and Kotlin practitioners display different tendencies when it comes to the contract type. In Java, there is a preference towards preconditions, while in Kotlin, postconditions are the most frequent type.

Although we can not provide a reason for this finding with certainty, analysing the most frequent constructs for pre and postconditions in both languages can give us some hints.

Tables 9 and 10 show the top 10 most frequent constructs per type in the last versions of Java and Kotlin apps, respectively. Comparing the two tables reveals distinct behavior patterns: for Kotlin, none of the top ten constructs relates to null-checking; however, for Java's instances reported in Table 9, 84.48% of preconditions and 73.05% of postconditions are associated with null-checking. In this number, we are not considering potential *IllegalArgumentException* and *IllegalStateException* that could be associated with null-checking since this would require analyzing the condition in the *if-statement*. This suggests a lack of expressiveness in the contracts specified by Java practitioners, with most being associated with null-checking, consistent with prior studies [33, 15].

This contrast in null-checking contracts between Java and Kotlin is easily explained by the languages' different takes on nullability. In Kotlin, regular types are non-nullable by default; therefore, in most cases, practitioners do not have the need for constructs like *AndroidXNonNull* or *JSR305NonNull*. On the other hand, it is interesting to observe that relaxing this constraint to allow nullable types is not a common practice since we found no meaningful use of constraints like *AndroidXNullable* and similar in Kotlin.

Table 9 The top 10 most frequent constructs per type in the last versions of Java applications.

Preconditions	Postconditions
AndroidXNonNull (45,399)	AndroidXNonNull (12,943)
AndroidXNullable (18,236)	AndroidXNullable (6,945)
IllegalArgumentException (7,663)	AndroidSuppressLint (3,125)
IllegalStateException (3,232)	AndroidTargetApi (1,243)
NullPointerException (2,230)	AndroidXRequiresApi (760)
GuavaPreconditionNotNull (1,021)	AndroidXWorkerThread (568)
AndroidXStringRes (1,008)	AndroidXCheckResult (474)
JSR305NonNull (860)	AndroidXCallSuper (421)
IndexOutOfBoundsException (656)	AndroidXKeep (398)
JetBrainsNotNull (612)	AndroidXUiThread (347)

Table 10 The top 10 most frequent constructs per type in the last versions of Kotlin applications.

Preconditions	Postconditions
AndroidXStringRes (1,162)	AndroidSuppressLint (2,289)
IllegalStateException (772)	AndroidXVisibleForTesting (1,663)
IllegalArgumentException (748)	AndroidXRequiresApi (738)
AndroidXColorInt (532)	AndroidXWorkerThread (638)
AndroidXDrawableRes (435)	AndroidXMainThread (442)
AndroidXAttrRes (255)	AndroidXCallSuper (323)
AndroidXColorRes (199)	AndroidXColorInt (244)
AndroidXIdRes (187)	AndroidTargetApi (205)
ProviderMismatchException (177)	AndroidXUiThread (196)
UnsupportedOperationException (116)	AndroidXAnyThread (184)

Finding 5: In Java applications, at least 80.85% of preconditions, 63.84% of postconditions, and 62.73% of class invariants are related to null-checking. In the case of Kotlin, we found only about 3.18% of preconditions, 7.17% of postconditions, and 0.66% of class invariants to be performing null-checking.

5.2 RQ2: First-to-Last Version Evolution

Table 11 presents the number of contracts in both versions by category. The *Type* column presents all types that are supported. In general, for most cases, the number of contracts in each category increased from the first to the last version. The only category where the number decreased was the *Apache's Commons Validate* for Java.

We computed some metrics to understand how the increase in the program's size relates to the number of contracts (see Table 12). These include the average and median values for the number of methods, the number of contracts, and the ratio between both (for both versions). The table shows that there is an average increase of about 114.185 methods per program. This is expected since the program's size tends to increase from the first to the second version. However, a more interesting insight comes from the contracts count. Although the average number of contracts per program increased, its median value decreased. This means that the dataset includes outliers with a significant rise in contract usage that considerably affected the average value. To confirm this data, we computed the ratio between

Table 11 Contract elements by type in both versions.

		contracts (1st vers.)		contracts (2nd vers.)	
Type	category	Java	Kotlin	Java	Kotlin
cond. runtime exc.	CRE	10,678	1,161	14,887	2,071
unsupp. op. exc.	CRE	203	26	308	116
java assert	assertion	1,308	-	2,217	-
kotlin assert	assertion	-	1,498	-	2,370
guava precond.	API	677	1	1,121	9
commons validate	API	11	0	3	0
spring assert	API	0	0	1	0
JSR303, JSR349	annotation	0	0	0	0
JSR305	annotation	2,062	7	2,133	13
findbugs	annotation	0	0	0	0
jetbrains	annotation	714	40	1,596	98
android	annotation	4,990	2,290	7,013	3,414
androidx	annotation	70,821	7,022	105,119	13,965
kotlin contracts	others	-	0	-	1

Table 12 Average and median number of methods, contracts, and their ratio for the two versions.

	1st version		2nd version	
Metric	Median	Average	Median	Average
methods count	288	925.175	334	1039.360
contracts count	8	72.567	7	86.807
contract-to-method ratio	0.038	0.072	0.030	0.064

the number of contracts and the number of methods for each version of a program. Then, we computed the difference between the second and the first version's ratio for each program. The average of these differences is -0.0077, and the median is -0.0012. Although the values are very small, we conclude that the number of methods increases significantly more than the number of contracts.

Finding 6: Apps that use contracts continue to use them in later versions. Moreover, the total and average numbers of contracts increase, but its median decreases by a small factor. Also, the number of methods increases at a higher rate than the number of contracts.

Similarly to our study, Dietrich et al. [13] also found that the median value of the ratio does not change much. Still, while we observed a decline between the two versions (from 0.038 to 0.030), they reported an increase (from 0.021 to 0.023). This means that although both studies show general stability related to contracts usage, contrary to their study, we were not able to find a positive correlation between the increase in the number of methods and in the number of contracts.

5.3 RQ3: Safety

To address whether practitioners tend to misuse contracts in either program evolution or inheritance contexts, we build *diff records* to be classified according to *evolution patterns*. Some of these *evolution patterns* are associated with a potential risk that may lead to client

```
-   @NotNull
    public Intent getIntent() { return intent; }
```

Listing 2 Example of a postcondition weakened using a Jetbrains annotation, taken from the project mGerrit, a Gerrit client for Android (in class SyncProcessor).

breaks, namely when *preconditions are strengthened* or *postconditions are weakened*. This process was described in more detail in Sections 4.3.3 and 4.3.4. It is important to note that the analysis tool cannot precisely capture all contract changes due to the variety of constructs we are analyzing and the complexity of their semantics. This can potentially lead to under-reporting. Another factor that may contribute to under-reporting is file path changes between versions, which may lead to no evolution patterns being detected. Even so, Table 13 still provides valuable insights into the safety of contract usage and evolution. The table shows the frequency of each *evolution pattern* in the context of *program evolution* (third column). We see that many contracts remain unchanged and that most changes are not critical. However, most of the changes that occur can lead to potential breaks, with *precondition strengthening* being over three and a half times more prevalent than *postcondition weakening*. An example of a precondition strengthening using an annotation and taken from our dataset was already shown in Listing 1. The code is from the class `ToolbarContentTintHelper` in project Retro Music Player,[12] a music player for Android. Adding `@NonNull` to the toolbar parameter strengthens the precondition by explicitly requiring callers to pass a non-null `Toolbar` instance, potentially breaking clients that previously relied on more permissive behavior. Listing 2 shows an example of a postcondition weakening. The code is taken from class `SyncProcessor` in project mGerrit,[13] a Gerrit client for Android. The postcondition is weakened because the `@NotNull` annotation promises a non-null `Intent`, but if `intent` is ever null, this contract is violated — potentially leading to runtime errors like NullPointerException in callers that rely on the non-null guarantee.

> **Finding 7:** There are instances of unsafe contract changes while the program evolves, particularly cases of preconditions strengthening.

Finally, Table 13 also presents the results found for *evolution patterns* in the context of *inheritance* (fourth column). We observe that the *precondition strengthening* pattern makes up almost 50% of classified instances. We also note that from the classified instances, most parts are related to contract changes which means a lack of stability in specifications. Both in the *evolution* and the *inheritance* study, we found lower occurrences of *postcondition weakening* when compared to the other classifications. Also, compared to the reports from Dietrich et al.'s study [13], our results indicate a greater ratio of *precondition strengthening* per preconditions found.

> **Finding 8:** There are instances of unsafe contract changes in an overriding context that violate the Liskov Substitution Principle, particularly cases of preconditions strengthening.

[12] https://github.com/RetroMusicPlayer/RetroMusicPlayer (last accessed on 01 April 2025)
[13] https://github.com/JBirdVegas/external_jbirdvegas_mGerrit (last accessed on 01 April 2025)

Table 13 Contract evolution in the context of program evolution and inheritance.

Contract Evolution	Critical	Evolution (#)	Inheritance (#)
unchanged	no	28,723	207
minor change	no	61	5
preconditions weakened	no	688	5
postconditions strengthened	no	1,035	76
preconditions strengthened	yes	1,963	284
postconditions weakened	yes	552	1
unclassified	?	858	159

6 Discussion

In this section, we answer the research questions listed in Section 4.1, we discuss the practical implications of our findings, and we outline threats to the validity of our work.

6.1 Answers to Research Questions

Based on our findings, we answer the research questions posed in Section 4.1 as follows:

RQ1 [Contract Usage] *How and to what extent are contracts used in Android applications?* Contracts are concentrated in a small number of apps. When applications use contracts, annotation-based approaches are the most frequent, with the *androidx.annotation* package being the most popular. The use of APIs to specify contracts is rare. While in Java, 64.80% of the classified instances are preconditions, Kotlin programs display a more equally distributed selection with 22.87% postconditions at the top. We also found that more than 60% of the classified contracts in Java are related to null-checking, while in Kotlin that number is smaller than 8%.

RQ2 [First-to-Last Version Evolution] *How does contract usage evolve in an application from the first to the last version?* Applications that use contracts continue to use them in later versions. When comparing the number of contracts in both versions, the average number of contracts increases. This is caused by some outliers that increase its usage substantially, driving up the average. In fact, the median value decreases. Furthermore, the contract-to-method ratio decreases between versions — an average decrease of -0.0077 and a median decrease of -0.0012. Although by a residual factor, we observed that the number of contracts declines as programs grow.

RQ3 [Safety] *Are contracts used safely in the context of program evolution and inheritance?* Contract changes are frequent and can lead to potential breaks, with *preconditions strengthening* being the most classified pattern. These results show a potentially unsafe use of contracts that may lead to client breaks and violate the Liskov Substitution Principle.

6.2 Practical Implications & Recommendations

Our findings lead to the following practical implications and recommendations.

Recommendation 1: Due to the fragmentation of technologies and approaches to specifying contracts, both Java and Kotlin standard libraries should be equipped with constructs to specify contracts and with proper official documentation.

Recommendation 2: It would be desirable to have libraries that standardize contract specifications in Java and Kotlin. Our results suggest that these libraries should be built around annotation-based contracts, given its popularity among practitioners. An annotation-based approach, where specifications are added to the program as metadata, is similar to Eiffel's approach, where the assertions do not obfuscate the method's implementation. This recommendation also applies to tool builders: given that the current use of APIs in Android development appears to be relatively low, analysis tools for Android that leverage contracts should prioritize annotations.

Recommendation 3: New tools to aid practitioners writing contracts would be valuable. For example, the integration into IDEs of contract suggestion features supported by tools for invariant inference, such as Daikon [14], could help increase practitioners' use of contracts. Another contribution could be IDE and continuous integration plugins to detect contract violations in the context of program evolution and inheritance.

Recommendation 4: Our findings show that Kotlin's default non-nullable types reduce the need to explicitly write some contracts, highlighting the significance of language design features that enable safety by default. These findings are relevant for the design of programming languages and can serve as motivation for practitioners when selecting programming languages for new projects.

6.3 User Study

To evaluate the recommendations we derived from our findings, and to gather challenges faced by practitioners when using contracts, we conducted a qualitative survey study with 16 practitioners. In particular, we are interested in answering the following research questions (SRQs):

- **SRQ1**. [**Challenges**] What are the main challenges that users face when using contracts?
- **SRQ2**. [**Recommendations**] What do users recommend to increase contracts' adoption?

6.3.1 Methodology

To answer our RQs, we designed a qualitative survey study.

6.3.1.1 Recruitment.

To improve the external validity, we allowed the participation of all kinds of software developers, but we recorded their experience with Android development. We recruited participants through Discord, LinkedIn, and our network (e.g., past students and colleagues). We also used snowball sampling by asking our contacts to distribute the study to their professional network. Our survey was implemented on Qualtrics and shared online. To prevent bots, all participants had to complete a reCAPTCHA challenge[14]. Per our inclusion criteria, participants were required to be at least 18 years old, in the United States, fluent in English, and possess some programming experience to ensure familiarity with basic software development concepts. All participants that we were able to recruit and who met the eligibility criteria were included in the final sample. Before deploying the study, we piloted it with five participants, iterating the survey between participants.

[14] reCAPTCHA is a security service provided by Google that protects websites from fraud and abuse by distinguishing human users from automated software.

6.3.1.2 Survey Description.

We begin our survey by showing participants the consent form. If they agree, we show the first section of our study, where we ask participants about their programming background and years of programming experience. Then, to ensure all participants are aware of the concept of DbC, we provide a short description and an example (see Figure 1). Participants are then asked about their confidence in understanding the definition of a contract, followed by questions regarding their frequency of contract use.

Design by Contract is a technique in which software systems are seen as components that interact amongst themselves based on precisely defined specifications of client-supplier obligations (contracts).
Contracts are specifications of an agreement between the client and the supplier of a component, where the supplier expects that certain conditions are met by the client before using the component (preconditions), maintains certain properties from entry to the component to exit (invariants), and guarantees that certain properties are met upon exit (postconditions). These contracts can be written as assertions directly into the code.
For example, a way of enforcing a precondition in Java using exceptions might be:

```
public void proceedWithCheckout ( List < Item> shoppingCart ) {
if ( shoppingCart.isEmpty () ) {
throw new IllegalArgumentException () ;
}
...
}
```

Other examples include annotations such as `@NonNull` , which can be used to express preconditions. In Java and Kotlin, the assert keyword can be used to enforce the validity of a condition (for example, an invariant). APIs such as `org.apache.commons.lang.Validate.*` or `com.google.common.base.Preconditions.*` are also used to denote contracts. Finally, Kotlin offers features such as `@ExperimentalContracts` that allow the developer to state a method's behavior to the compiler explicitly.

Figure 1 Explanation shown to participants about DbC.

Here, the survey is split into two parts. For those who never use contracts, a follow-up section asks for the reasons for not using contracts. Participants who use contracts are asked to describe their reasons for using contracts and any challenges they have encountered. This is followed by the recommendations section. It begins by asking participants to suggest ways to improve the adoption of contracts. Following this, participants are presented with the following recommendations to improve contracts, obtained from the findings of our empirical study:

- Extend Java and Kotlin standard libraries with specialized constructs to specify contracts and with proper official documentation.
- Have libraries that standardize contract specifications in Java and Kotlin.
- Integrate into IDEs contract suggestion features supported by tools that automatically generate assertions and contracts.
- IDE and continuous integration plugins to automatically detect contract violations.

Participants were asked to rank these recommendations in terms of importance. Finally, the survey concludes with a demographic section.

The recommendations presented to participants in the user study were derived from our empirical findings but reformulated in a more concise and direct way. Presenting the recommendations exactly as shown in Section 6.2, which includes both context and the recommendation itself, was deemed too verbose for the user study.

6.3.1.3 Ethical Considerations.

The study was approved by the IRB of Carnegie Mellon University. The participants did not receive payment upon survey completion. All participants were shown a consent form before filling in the survey. We did not collect any personally identifiable data.

6.3.1.4 Demographics.

We recruited two participants for the initial pilot and three more for the follow-up pilot. For the finished survey, we recruited 23 participants. Of those 23, seven were ineligible or did not pass our screening questions (e.g., by not having programming experience). The remaining 16 participants sample is composed of individuals aged between 18 and 44 years, with most (nine participants) in the 25-34 age bracket. Gender representation includes male, female, non-binary/third gender, and one participant preferring not to disclose their gender. Educational backgrounds are high, with most participants holding graduate or professional degrees and a smaller portion possessing bachelor's degrees. The sample is primarily White or Caucasian, with one Asian participant and one preferring not to disclose their race. Programming experience among the participants is diverse, with Python being the most commonly used language, followed by Java, JavaScript, C++, Rust, TypeScript, Go, C, Kotlin, and Dafny. All participants had some programming experience, with five participants having 1-3 years, three with 4-6 years, another five with 7-10 years, and finally, two with over 10 years of experience. Only one had less than one year of experience. Regarding experience with Android development, about half of the participants, 9 out of 16, had no years of experience. A subset had some experience, with one participant having between 1-3 years and another 7-10 years. The remaining five participants had less than one year of experience with Android software development.

6.3.1.5 Analysis

We used descriptive statistics to analyze the survey data from the closed-answer questions. For the qualitative responses, we developed three distinct codebooks tailored to different aspects of the dataset: 1) the reasons behind participants' use or non-use of contracts, 2) the challenges encountered while using contracts, and 3) the recommendations offered by participants to enhance the adoption of contracts. We used emergent coding techniques to develop the codebooks. We followed an iterative process to code the qualitative data. One of the researchers began by creating the first versions of the three codebooks. After this, two researchers independently double-coded all the answers, refined the codebook, recoded the answers again, and finally met to discuss any disagreements and reach a consensus.

6.3.2 Results

Among our survey participants, all, except one, reported using contracts in their programming practices, citing various reasons that underscore the multifaceted benefits of this approach. The participant who said they did not use contracts attributed their decision to the informal nature of their programming work, mainly prototyping and scripting. A significant majority, 11, highlighted the role of contracts in enhancing code quality and reliability. They mention that they use contracts to assert postconditions, verify preconditions, detect bugs, and identify edge-case bugs. This ensures that the code behaves as expected across compile-time and runtime scenarios. Four participants mentioned the importance of contracts as a documentation tool for improving code clarity. Three responses said that they used contracts in software design to manage expectations for software behavior. Lastly, two participants pointed out the operational benefits of contracts in enhancing the development process. They mentioned how contracts facilitate "sanity checks" (Participant 10) and ensure compliance with requirements.

Table 14 Codebook for participants' challenges when using contracts.

#	Code	Description
3	Maintenance and Flexibility	Problem with maintenance of contracts when implementations change, and the perceived lack of flexibility with contracts.
2	Specification and Expressiveness	Challenges in defining specifications and on the balance between contract expressiveness and automatic verification capabilities.
2	Cognitive Overload and Integration	Increased cognitive load due to managing both code and contracts, and integrating contracts into existing codebases.
2	Loop Invariants and Abstraction Levels	Specific challenges in formulating loop invariants and choosing the appropriate level of abstraction.
2	Enforcement Challenges	Challenges related to effectively enforcing contracts within the development process.
1	Security Concerns	Potential security risks.
1	Learning Curve and Documentation	Initial learning curve, difficulty in understanding contract libraries and navigating the documentation.

6.3.2.1 SRQ1: Challenges

This subsection addresses SRQ1 and explores users' main challenges when using contracts in software development.

Participants provided diverse answers when questioned about their challenges when using contracts. In Table 14, we describe the codes and their respective frequency in participants' answers. The most cited obstacle was *Maintenance and Flexibility*, mentioned by three participants. This code highlights the sometimes complicated tasks of maintaining and updating contracts in complex projects. Participant 12 mentioned, "*if the implementation changes, we need to update the contract, and so, it can become complex to know which contracts need to be updated*". Challenges like *Specification and Expressiveness, Cognitive Overload, Loop Invariants*, and *Enforcement* were each present in the answers of two participants. And, finally, *Security Risks* and *Learning Curve and Documentation* were mentioned as challenges by one participant each.

6.3.2.2 SRQ2: Recommendations

This subsection addresses SRQ2 and users' recommendations to improve the adoption and usage of contracts in software development.

As mentioned before, we showed participants four recommendations obtained from our empirical study. Overall, participants seem to value all recommendations previously identified, as most classify them as "Very Important" and "Somewhat Important". The recommendation that participants seem to value the most is "IDE and continuous integration plugins to automatically detect contract violations" with 14 saying it is "Very Important" for them and two "Somewhat Important". This recommendation is closely followed by the one that suggests integrating contracts into IDEs ("Integrate into IDEs contract suggestion features supported by tools that automatically generate assertions and contracts") with 11 saying it is "Very Important" for them, and five "Somewhat Important." The remaining suggestions are to extend standard libraries with specialized constructs to specify contracts and with proper official documentation; these were also valued by participants, but one participant showed some uncertainty and indicated they were "Not sure" and classified it as "Not Important at All". Our results suggest that participants view the recommendations identified in our

Table 15 Codebook for participants' suggestions, including the frequency and description of each code.

#	Code	Description
7	Tool Support and Integration	Developing tools and IDE integrations that assist in creating, verifying, and managing contracts.
3	Educational Resources and Training	Providing more educational materials, examples, and training on DbC principles and benefits.
3	Error Handling and Debugging Support	Ensuring error recovery mechanisms and developing tools to simplify debugging processes related to contract violations.
2	Standards and Guidelines	Establishing standards or guidelines for how contracts should be defined, including preconditions and postconditions.
2	Incremental Adoption Strategies	Encouraging incremental adoption of DbC to make it easier for developers to integrate into their workflows.
2	User Interface and Templates	Providing user interfaces and templates to facilitate the writing of contracts and automatic code generation/repair.
2	NLP and AI	Utilizing NLP and AI for contract code suggestions.
2	Specification / Code Repair	Providing the ability to repair code based on changes to specifications (contracts) or update specifications based on code changes.
1	Programming Language Support	Enhancing programming language features to support contracts more effectively.
1	Automatic Verification and Testing	Improving automatic verification of contracts with less human effort and generating tests from contracts.
1	Real-Time Feedback and Metrics	Integrating real-time feedback and metrics within IDEs to provide indicators of code quality and contract coverage..

empirical work as valuable and support our insights.

Before asking participants to rank the previously identified recommendations, we asked them to suggest ways to improve the adoption of DbC. The codebook with the frequency of each code can be seen in Table 15. Participants' answers were diverse and seemed to also validate our results. The most frequent code in participants' suggestions is *Tool Support and Integration*: in total, seven participants suggested developing tools and IDE integrations that assist in creating, verifying, and managing contracts. This code validates our findings as it is similar to the recommendations that we derived from our empirical study. The second most frequent codes were the ones related to providing educational materials, templates, user-friendly interfaces, and robust error handling for users. The codes *Educational Resources and Training, Error Handling and Debugging Support, and User Interface and Templates* were each found three times in participants' answers. These recommendations suggest that participants need resources that support them in the practical implementation of contracts. Participant 9 directly says that they think that a way to improve the adoption of contracts is to "*always make sure there is a way to recover from the exceptions thrown whenever the assert (Python) statement is used.*" *Standards and Guidelines, Incremental Adoption Strategies, Natural Language Processing and AI, and Specification / Code Repair* were each mentioned twice. Particularly, *Natural Language Processing and AI* in similar ways by two participants, with Participant 7 saying "*I think AI contract code suggestions would reduce the barrier to entry and cost of writing the code.*" Finally, *Programming Language Support, Automatic Verification and Testing, and Real-Time Feedback and Metrics* were mentioned once, reinforcing that participants desire more automatic implementations of contracts and more feedback from their application.

Overall, our results suggest a clear direction — developers seem to desire improved tool support and integration of DbC in the development process. Our results highlight the need for future work on contracts and validate the findings of our empirical study.

6.4 Threats to Validity

Internal Validity. The accuracy of our results depends on the quality and correctness of the artifact, and there may exist bugs in the code. To mitigate this, we extensively tested the tool. In addition, all code and datasets used are publicly available for other researchers and potential users to check the validity of the results. Regarding the user study, one potential threat is the Hawthorne effect, where participants may alter their behaviour because they are aware they are being observed. To mitigate this risk, we ensured that participation was confidential and that responses could not be linked to individuals. ***External Validity.*** The projects that we selected might not be an accurate representation of other, more popular, Android app stores. We mitigated this by using F-Droid, a collection of open-source applications commonly used in other research studies. We also mitigated this risk by analysing *all* the projects that satisfy the inclusion criteria, leading to a substantial dataset (51 MLoC) with applications of different types. Regarding the user study, one potential threat arises from the fact that about half of the participants lacked prior experience with Android development. As a result, the findings may not fully generalize. ***Conclusion Validity.*** We might have missed language constructs that could be used to specify contracts. To mitigate this, we followed an established taxonomy [13] that we adapted and extended by systematically searching forums and the official Android documentation. The full list of constructs is available in the Supplementary Material [17]. Also, all our code is easily open to extension. Another risk comes from our dataset being imbalanced (with more Java than Kotlin applications). We mitigate this by explicitly discussing this imbalance when presenting results that might be affected by it.

7 Conclusions

Empirical evidence about contract usage can help the software engineering community create or improve existing libraries and tools to increase DbC adoption. This also helps to understand DbC's current practices better, helping practitioners discover and decide between different approaches. Researchers can also use our contributions to conduct additional studies.

Future work includes large-scale studies with practitioners to understand the challenges faced when specifying contracts, the use of annotations to improve Android analysis tools [24, 32, 30, 31], and the development of tools that can help increase the adoption of DbC [20, 43, 2].

References

1 Y. A.Feldman, O. Barzilay, and S. Tyszberowicz. Jose: aspects for design by contract. In *Fourth IEEE International Conference on Software Engineering and Formal Methods*, Los Alamitos, CA, USA, 2006.

2 Shibbir Ahmed, Sayem Mohammad Imtiaz, Samantha Syeda Khairunnesa, Breno Dantas Cruz, and Hridesh Rajan. Design by contract for deep learning apis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 94–106, 2023.

3 A. Algarni and K. Magel. Toward design-by-contract based generative tool for object-oriented system. In *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS). Proceedings*, pages 168 – 73, Piscataway, NJ, USA, 2018.

4 M. Aniche. *Effective Software Testing. A Developer's Guide.* Manning, Shelter Islands, 2022.

5 M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 356–367. Association for Computing Machinery, 2016.

6 Joshua Bloch. *Effective java.* Addison-Wesley Professional, 2nd edition, 2008.

7 M. Blom, E. J. Nordby, and A. Brunstrom. On the relation between design contracts and errors: a software development strategy. In *Proceedings Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 110–117, 2002.

8 M. Blom, E.J. Nordby, and A. Brunstrom. An experimental evaluation of programming by contract. In *Proceedings Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 118–127, 2002.

9 C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). Proceedings*, volume 1, pages 755 – 66, Los Alamitos, CA, USA, 2015.

10 P. Chalin. *Are practitioners writing contracts?*, pages 100 – 113. Springer, Berlin, Germany, 2006.

11 Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 596–607. IEEE, 2019.

12 S. Counsell, T. Hall, T. Shippey, D. Bowes, A. Tahir, and S. MacDonell. Assert use and defectiveness in industrial code. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops*, pages 20–23, 10 2017.

13 J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada. Contracts in the wild: A study of java programs. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

14 Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

15 H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer. Contracts in practice. In *FM 2014: Formal Methods. 19th International Symposium. Proceedings: LNCS 8442*, pages 230 – 46, Cham, Switzerland, 2014.

16 G. Fairbanks. Better code reviews with design by contract. *IEEE Software*, 36(6):53 – 6, 2019.

17 David R. Ferreira, Alexandra Mendes, João F. Ferreira, and Carolina Carreira. Contract usage and evolution in Android mobile applications (supplementary material), 2025. Available online at: `https://archimendes.com/publication/2025/ecoop/ecoop25-AndroidContracts-SupplementaryMaterial.pdf`.

18 L. Di Grazia and M. Pradel. The evolution of type annotations in python: An empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 209–220, New York, NY, USA, 2022. Association for Computing Machinery.

19 B. Hollunder, M. Herrmann, and A. Hülzenbecher. Design by contract for web services: Architecture, guidelines, and mappings. In *International Journal on Advances in Software*, volume 5, 2012.

20 Marieke Huisman and Raúl E Monti. Teaching design by contract using snap! In *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, pages 243–263. Springer, 2022.

21 P. Kochhar and D. Lo. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 298–307, 2017.

22 Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*, pages 204–212, 2006.

23 Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 438–454, 2024.

24 Olivier Le Goaer and Julien Hertout. Ecocode: A sonarqube plugin to remove energy smells from android projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

25 K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

26 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40 – 51, 1992.

27 Bertrand Meyer. Programming as contracting. *Advances in Object-Oriented Software Engineering*, pages 1–15, 1988.

28 P. V. R. Murthy. Design by contract methodology. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 482 – 8, Piscataway, NJ, USA, 2018.

29 A. Naumchev. Seamless object-oriented requirements. In *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON). Proceedings*, Piscataway, NJ, USA, 2019.

30 Ricardo B Pereira, João F. Ferreira, Alexandra Mendes, and Rui Abreu. Extending Ecoandroid with automated detection of resource leaks. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pages 17–27, 2022.

31 Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. Ecoandroid: An Android studio plugin for developing energy-efficient Java mobile applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 62–69. IEEE, 2021.

32 Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1232–1244, 2022.

33 T. W. Schiller, K. Donohue, F. Coward, and M. D. Ernst. Case studies and tools for contract specifications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 596–607, New York, NY, USA, 2014. Association for Computing Machinery.

34 C. Silva, S. Guerin, R. Mazo, and J. Champeau. Contract-based design patterns: a design by contract approach to specify security patterns. In *ARES 2020: Proceedings of the 15th International Conference on Availability, Reliability and Security*, New York, NY, USA, 2020.

35 StatCounter Global Stats. Operating system market share worldwide, 2024. [Online; accessed 01-April-2025]. URL: `https://gs.statcounter.com/os-market-share#monthly-202411-202412-bar`.

36 J. Tantivongsathaporn and D. Stearns. An experience with design by contract. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 327 – 33, Piscataway, NJ, USA, 2006.

37 K. Tao and P. Edmunds. Mobile apps and global markets. *Theoretical Economics Letters*, 08:1510–1524, 01 2018.

38 Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE*

International Conference on Software Maintenance and Evolution (ICSME), pages 35–45, 2020.

39 Y. Wei, C.A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *2011 33rd International Conference on Software Engineering (ICSE 2011)*, pages 191 – 200, Piscataway, NJ, USA, 2011.

40 Z. Yu, C. Bai, L. Seinturier, and M. Monperrus. Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, 47(5):969–986, 2021.

41 Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24:3394–3434, 2019.

42 Y. Zhou, P. Pelliccione, J. Haraldsson, and M. Islam. Improving robustness of autosar software components with design by contract: A study within volvo ab. In *Software Engineering for Resilient Systems. 9th International Workshop, SERENE 2017. Proceedings: LNCS 10479*, pages 151 – 68, Cham, Switzerland, 2017.

43 Álvaro Silva, Alexandra Mendes, and João F. Ferreira. Leveraging large language models to boost Dafny's developers productivity. In *International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2024. `arXiv:2401.00963`.