FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Studying Contract Usage in Android Mobile Applications

David Regatia Ferreira

Master in Informatics and Computing Engineering

Supervisor: Prof. Alexandra Mendes

Second Supervisor: Prof. João F. Ferreira

July 20, 2023

Studying Contract Usage in Android Mobile Applications

David Regatia Ferreira

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

July 20, 2023

Abstract

The success of a software system is highly correlated to its reliability, which is a growing concern for mobile applications due to their ubiquitous nature and large user base. In this context, Design-by-Contract proposes a systematic specification of contracts that define the interaction between components to build more robust and correct software. However, the few existing studies suggest a low usage of this methodology by practitioners. Although its theoretical advantages are widely discussed in the literature, the lack of empirical evidence about its benefits and the absence of standard patterns and tools is considered to be the main reason for the industry's skeptical view on its adoption.

This research is the first large-scale empirical study on the presence and use of contracts in Android applications. By evaluating approximately 4,000 applications, it is possible to derive recommendations for practitioners, researchers, and tool builders. More specifically, this study considers applications written in Java or Kotlin and explores, among other questions, (i) how and to what extent contracts are used in Android applications, (ii) how contract usage evolves in an application, and (iii) whether contracts are used safely in the context of program evolution and inheritance. The results show that (i) although most applications do not specify contracts, annotation-based approaches are the most popular among practitioners , (ii) was not found any positive correlation between the number of contracts and the increase in the program's size, and (iii) there are many potentially unsafe specification changes when the program evolves and in subtyping relationships.

These results show the necessity for additional efforts to provide more specialized libraries for contract specification, including in the Java and Kotlin languages. At the same time, it is also necessary to reinforce the dissemination and teaching of this software design methodology to close the gap between the industry and academia.

Keywords: design by contract, android applications, assertions, verification, software reliability

ACM Classification: General and reference → Cross-computing tools and techniques → Empirical studies; General and reference → Cross-computing tools and techniques → Reliability.

Resumo

O sucesso de um sistema de software está altamente correlato com a sua fiabilidade. Esta é uma preocupação crescente em aplicações móveis devido à sua natureza ubíqua e ao seu grande número de utilizadores. Neste contexto, Design-by-Contract defende a especificação sistemática de contratos que definem a interação entre componentes para construir software mais robusto e correto. Contudo, os poucos estudos existentes sugerem uma baixa utilização desta metodologia por profissionais. Apesar das vantagens teóricas serem amplamente discutidas na literatura, a falta de evidência empírica sobre os seus benefícios e a carência por padrões e bibliotecas *standards* são consideradas as principais razões pela visão cética da indústria relativamente à sua adoção.

Este trabalho é o primeiro estudo empírico de larga escala sobre a utilização e a presença de contratos em aplicações Android. Deste modo, ao analisar cerca de 4,000 aplicações, é possível extrair um conjunto de recomendações para outros profissionais, investigadores e criadores de ferramentas e bibliotecas. Mais especificamente, este estudo considera aplicações escritas em Java e Kotlin e explora, entre outras questões, (i) em que medida e de que forma contratos são utilizados em aplicações Android, (ii) de que forma os contratos evoluem numa aplicação e (iii) se contratos são utilizados de forma segura no contexto de evolução do programa e de herança. Os resultados mostram que (i) apesar da maioria dos programas não usarem contratos, representações baseadas em anotações são as mais populares entre profissionais, (ii) não foi encontrada qualquer correlação positiva entre o número de contratos e o aumento do tamanho da aplicação, e (iii) existem várias ocorrências de alterações de especificações potencialmente não seguras quando o programa evolui e no contexto de relações de *subtyping*.

Estes resultados mostram a necessidade por esforços em construir bibliotecas e ferramentas mais especializadas para especificação de contratos, incluindo nas linguagens Java e Kotlin. Ao mesmo tempo, é também necessário reforçar a disseminação e ensino desta metodologia de design de software para aproximar a indústria da academia.

Palavras-Chave: design by contract, aplicações android, asserções, verificação, fiabilidade do software

Classificação ACM: General and reference → Cross-computing tools and techniques → Empirical studies; General and reference → Cross-computing tools and techniques → Reliability.

Acknowledgements

Firstly, I want to express my deep gratitude towards my advisors, Professor Alexandra Mendes and Professor João F. Ferreira, for guiding me through this work and introducing me to the empirical research process.

Additionally, I want to thank my family for all the support throughout my life, especially my parents. Their 2006 summer decision to move the family thousands of kilometers to ensure that my brother and I would one day attend university was crucial for this moment.

Lastly, I want to share my gratitude to my girlfriend, Francisca, for being with me since the beginning of this journey and for encouraging me to do my best. You have been my rock for the last seven years and many more to come.

David Regatia Ferreira

"For I saw with my own eyes sights
Which rough sailors, whose only schooling
Is observation and long experience,
Take as knowledge, evident and sure,
And which those with higher intelligence
Who use their skills and learning
To penetrate earth's secrets (if they could),
Dismiss as false or feebly understood"

Luís de Camões, The Lusiads, Canto V

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application programming interface
AST	Abstract syntax tree
CRE	Conditional runtime exception
DbC	Design-By-Contract
JVM	Java virtual machine
LSP	Liskov Substitution Principle
OOP	Object-oriented programming

Chapter 1

Introduction

This chapter provides an introduction and an overview of the problem under study. Section 1.1 presents the context of the problem and its main concepts. Section 1.2 addresses the motivation that leads to this work. Section 1.3 defines the problem under investigation, its research questions, and its main contributions. Finally, Section 1.4 presents the structure of this document.

1.1 Context

Software is increasingly ubiquitous in our daily life, and its complexity is ever-increasing. This is both an opportunity — to advance human well-being and extend its capabilities — and a challenge — a greater dependency on software and its correct functioning. Since software is becoming more embedded in our day-to-day life, considering reliability while designing and developing software is crucial for today's society.

According to Meyer (1997), software reliability encompasses correctness — the capability to accomplish tasks as defined by the specification — and robustness — the flexibility to react correctly to abnormal conditions. In this context, software components are often seen as the weakest link when addressing the reliability of a system (Murthy, 2018). Hence, to address both quality factors, and supported by earlier works in formal verification, formal specification, and Hoare logic (Naur, 1966; Hoare, 1969; Floyd, 1993), Bertrand Meyer consolidated the Design-by-Contract (DbC) methodology, which advocates for formal and precise specifications, *contracts*, in software components to define the interaction between clients and suppliers and to assert each party's rights and obligations. Those specifications can are expressed as *pre-conditions*, *post-conditions*, or *class invariants*. A pre-condition defines the properties that must hold when a method is invoked. A post-condition states the properties that the method guarantees when it returns. Lastly, invariants are specifications associated with fields that must hold during the object's lifetime.

Generally, when a contract is broken, the program throws an exception, which helps detect errors early and facilitates debugging (Aniche, 2022). Additionally, they serve as living documentation by clearly stating the expectations of each module during their interaction which is fundamental in reusability patterns (Wei et al., 2011).

However, the contract specification support depends on each language, and most do not provide native support for Design-by-Contract. Still, as is the case for *Java* and *Kotlin* languages, there are some constructs like *asserts* and *exceptions* and even third-party libraries that allow developers to take advantage of this technique in their projects (Dietrich et al., 2017). Nonetheless, these multiple non-native approaches contribute to a fragmentary and non-standardized usage of DbC.

1.2 Motivation

The success of a software system is highly correlated to its reliability. This fact is even more apparent when considering critical systems whose failure could result in loss of life, significant property damage, or damage to the environment (Knight, 2002). For this reason, investigating techniques capable of improving software reliability, such as DbC, is particularly interesting to the industry, academia, and society in general.

Moreover, object-oriented programming is an industry standard and supports most software built today. This paradigm relies on dividing work between flexible, modular, and reusable software components that interact with each other. This requires particular caution when designing software so those values — reusability, extendibility, and compatibility — don't come at the cost of reliability (Meyer, 1997). Features usually found in those languages, such as inheritance, polymorphism, and dynamic binding, can also present additional challenges to designing correct and robust components (Xu and Xu, 2010).

This urgency to build reliable software is a growing concern for mobile applications. Many companies are adopting iOS and Android as target platforms for their apps in critical domains such as mobility, health, finance, and government. Additionally, according to Tao and Edmunds (2018), smartphone users have reached 2.5 billion by 2018, with over 2 million apps available on the App Store and Google Play. Additionally, data from 2022 shows that Android represents approximately 43% of the overall operative system market share, followed by Windows (29%), and then iOS (18%) (Stats, 2023). Therefore, faults in mobile apps, particularly Android apps, can impact millions of users. Hence, the importance of reliability techniques research for mobile applications.

Since the 1980s, many researchers have advocated Design-by-Contract as an efficient technique to aid the identification of failures (Aniche, 2022), improve code understanding (Fairbanks, 2019), and improve testing efforts (Tantivongsathaporn and Stearns, 2006) which directly or indirectly contribute to the improvement of reliability. Still, to our knowledge, there are no previous studies on the presence and usage of contracts in Android applications or any study that includes the Kotlin language. Even though there is some investigation on standard Java applications (Estler et al., 2014; Dietrich et al., 2017), as Jha and Nadi (2020) states, researchers need to distinguish mobile applications when performing studies related to programming language features due to their differences in nature, size, and libraries used. Therefore, the present work not only fills a gap in the empirical study of contracts but also represents a source of knowledge to compare how the adoption and usage of contracts in Android applications differ from standard Java applications.

After all, more empirical evidence about what types of constructs and language features practitioners use to represent contracts can help builders create or improve existing libraries and tools to increase DbC adoption. This knowledge also serves practitioners to understand DbC's current practices better, helping them discover and decide between different implementation approaches of this technique for their projects. Other researchers can also use this data to draw additional studies and to foster further discussion along with the increasing interest in these empirical studies about language features.

1.3 Problem Statement

Design-by-Contract has been receiving attention from academia since its formulation in the 1980s. Still, although a considerable amount of work presumes the advantages and limitations of the technique, there are not sufficient empirical studies on the usage of DbC by practitioners. More particularly, studies targeting Kotlin or Android applications are non-existent to our knowledge. This lack of information about how the industry uses contracts is an obstacle to the continuous creation and improvement of tools and libraries by researchers and builders (Schiller et al., 2014). Additionally, insufficient empirical evidence about the benefits of contracts hampers their adoption by practitioners and the establishment of DbC as a software design standard (Tantivongsathaporn and Stearns, 2006).

Therefore, this work aims to study the presence and usage of contracts in a large dataset of Android applications written in Java or Kotlin while also analyzing whether practitioners use contracts safely in the context of inheritance relations and while the project evolves. The generated empirical data leads to a more comprehensive view of the adoption rate and usage patterns among Android developers and the language features employed.

To support this study, an artifact is developed to execute an automated examination of contracts in Android applications and produce the empirical data necessary for this work's discussion. This artifact is written in Java and is based on the static code analysis tool proposed by Dietrich et al. (2017).

1.3.1 Research Questions

In this work, we address the following research questions (RQ):

RQ1. How and to what extent are contracts used in Android applications?

Are Android developers using contracts in their applications? What types of representations are used? Are pre-conditions, post-conditions, and invariants used equally?

RQ2. How does contract usage evolve in an application?

As time evolves, do applications that use contracts reinforce its usage? How does the number of contracts change between two versions of the same application?

RQ3. Are contracts used safely in the context of program evolution and inheritance? Are there instances where a contract evolution can lead to a client break? Do we find contracts that violate Liskov's Substitution Principle (LSP)?

1.3.2 Contributions

The main contribution of this work is to increase current knowledge about the usage and adoption of contracts by real-world applications and, more particularly, by Android applications. As mentioned earlier, more empirical data about contract adoption and usage in the industry can be valuable for practitioners, tool builders, and researchers. In particular, this study fills the missing data related to Android applications and the Kotlin language.

Additionally, as a result, this study provides an extension of the tool proposed by Dietrich et al. (2017) to analyze contracts in Java and Kotlin source code. In this process, this tool was refactored to facilitate the extension of its support to other languages.

In summary, the contributions of this dissertation are:

- Creation and categorization of a list of language features, tools, and libraries to represent contracts in Android applications.
- Creation of a pipeline that builds a large-scale dataset of Java and Kotlin open-source projects, including inclusion criteria and size optimization.
- Extension of the static analysis tool proposed by Dietrich et al. (2017) to analyze Kotlin code and to investigate additional Android-specific contracts.
- Reporting of empirical data about contract adoption and usage as a result of the study of a large-scale dataset of Android applications.
- A set of recommendations for practitioners, researchers, and tool builders about contract adoption and research.

1.4 Document Structure

This document is structured as follows:

- Chapter 2 introduces the reader to fundamental concepts and discussions related to the methodology under study, Design-By-Contract. It also summarizes related work, including empirical studies and their findings on contract, assertions, and annotations usage.
- Chapter 3 describes the proposed solution by explaining the dataset creation process and how the static analysis tool produces data that support each research question's discussion. This chapter also presents the contract constructs under study and how they are categorized.
- Chapter 4 presents the results and findings of the empirical study of a large-scale Android applications dataset.

- Finally, Chapter 5 presents final remarks about this work, describing its main results by listing a set of implications to practitioners, tool builders, and researchers.

Chapter 2

Background and Related Work

This chapter introduces fundamental concepts related to the technique under investigation, Design-by-Contract, and discusses related work. The section 2.1 defines software reliability and introduces the notion of robustness and correctness, which are essential to understand DbC. Then, section 2.2 explains the motivations and philosophy behind the technique and discusses its advantages and limitations. Last, Section 2.3 presents the existing empirical studies on contracts, asserts, and annotations usage.

2.1 Software Reliability

The definition of *software reliability* has been proposed by many authors with a general consensus. For instance, Myers (1976) states that reliability is the probability that the software will execute for a particular period of time without failures, considering the cost of each failure encountered. McConnell (2004) defines reliability as the ability of a system to perform its requested functions under stated conditions whenever required - having a long mean time between failures. Similarly, (Meyer, 1997) states that reliable software is software that is both *correct* - performs its exact tasks, as defined by the specification - and *robust* - reacts appropriately to abnormal conditions. Thus, in Meyer's definition, robustness complements correctness. While correctness ensures that the system does what it is supposed to do, robustness makes sure that when a case arises outside the specification, the system does not cause a catastrophic event. Due to Meyer's important role in Design-by-Contract, his definition of software reliability is central to our work.

Meyer's definition of *correctness* highlights another important idea: correctness is a relative notion. As the author states, a software system is neither correct nor incorrect by itself. We can only determine the correctness of a system when we compare it against a certain specification. Therefore, a correct software system is a system that is consistent with its specification. By implication, we can only build reliable software if we have a well-defined specification. As we'll see in the next sections, the role of *specifications* is central to the Design-by-Contract methodology.

2.2 Design-By-Contract Methodology

2.2.1 The Problem and Motivations

The Design-by-Contract (DbC) term was formulated by Bertrand Meyer while designing the Eiffel language. Eiffel was made public in 1986 and was conceived as a tool to create reliable commercial software by providing native support for contracts specification. Meyer's vision of contracts as lightweight specifications was an important milestone, but the notion of contracts dates earlier to the works of Robert Floyd, Tony Hoare, and Edgar Dijkstra on logical reasoning in procedural programming and formal specification and verification of software (Hoare, 1969; Floyd, 1993; Fairbanks, 2019).

New challenges to achieve reliable software systems were uncovered with the rise of object-oriented programming (OOP) and particularly with one of its core values, *reusability*. This paradigm is based on the concept of objects (and classes) that, while encapsulating data and behavior, interact with each other to perform a task. Programming languages that support this paradigm offer many mechanisms that allow the developer to make use of *encapsulation* and *inheritance* to enhance reusability. Still, as Jazequel and Meyer (1997) argues, the lack of a precise specification in a reusable module can lead to awful consequences, such as the ones that will be presented next.

A well-known example of this problem is the Ariane 5 Flight 501 explosion in 1996. The issue was caused by the software system when trying to convert an unexpectedly large value to a 16-bit signed *integer*. Since the value exceeded the range representable by 16 bits, an overflow error was thrown and not properly handled (Lann, 1997). In this case, we have a clear example of a robustness issue when the system was not able to "graciously" handle the unexpectedly large number generated for the horizontal bias. Additionally, there was a lack of a precise and visible specification that the horizontal bias should fit in 16-bit signed *integer*. In this specific example, the introduction of a precise specification in the reusable component could have led to the team finding the issue during the development or validation phase before the flight.

More recently, in 2014, a security vulnerability known as the "Heartbleed Security Vulnerability" was found in the OpenSSL library that provides an implementation of internet security protocols like SSL and TLS. In short, due to a missing security check, an attacker could send in a heartbeat request a payload length much larger than the actual record size, receiving copies from the actual payload and the memory content adjacent to it (Carvalho et al., 2014). Thus, attackers could easily retrieve passwords and private keys by exploiting this issue. Once again, the development and testing team did not detect a missing check while implementing and reviewing the code. This could have been avoided if a precise specification had been added to the code bringing attention to this behavior.

2.2.2 The Notion of Contract

In order to prevent problems like the ones described in the previous section, Bertrand Meyer advocates for the specification of contracts that regulates the interaction between software components.

As the author argues, since object-oriented programming is all about creating modular components that many other modules will reuse, an issue in robustness or correctness can quickly propagate, increasing its negative effect (Meyer, 1997). Therefore, with contracts, modules cooperate on the basis of precisely defined specifications avoiding the occurrence or the propagation of faulty behavior. Those contracts not only ensure the correct functioning at runtime but also serves as documentation for developers, code reviews, and testers. This is extremely important in critical systems and systems that deal with complex financial operations (Murthy, 2018; Aniche, 2022).

This idea of contracts is built on top of the very notion of contracts present in society. A contract is an agreement between at least two parties that defines mutual rights and obligations. Therefore, a contract protects both sides by clearly identifying which side is to blame if the contract is broken. As described in section 2.1, a specification, or contract, can help to measure the system's reliability and can be translated by a *correctness formula* of the form of

$$\{P\}A\{Q\}$$

which means that "any execution of A, starting in a state where P holds, will terminate in a state where Q holds". Hence, a contract between a client and a supplier in software can assume three forms: *pre-condition*, *post-condition*, or *class invariant*. Given the correctness formula, the P assertion is called a pre-condition and Q the post-condition.

A *pre-condition* specifies what needs to hold when the service is invoked. The client is bound to satisfy the requirements as stated by the pre-conditions, and the supplier is ensured that certain properties will be true after the routine call.

A *post-condition* specifies the state that must be ensured on return. It is the supplier's responsibility to perform its job as stated by the post-condition. It assures the client that certain properties will hold after the routine call.

In a more verbose explanation, a contract can be read as the supplier stating that "if you promise to call the *routine r* with the *pre-conditions* satisfied, then I, in return, promise to deliver a final state in which the *post-conditions* are satisfied" (Meyer, 1997).

A classic example of applied pre-conditions and post-conditions is considering a *push(x)* method that adds x to the end of a stack. Providing that the method is invoked by the client only when the stack is not full (pre-condition), the supplier promises to add x to the last position of the stack and to increase the counter by 1 (post-conditions). The client benefits from having the stack updated, and the supplier does not need to verify whether the stack is full, simplifying its processing.

Additionally, contracts can also be specified in the form of *class invariants*. Contrary to pre-conditions and post-conditions, which are associated with methods, class invariants are defined for classes' properties to ensure the consistency of those properties and, by implication, the consistency of the object itself between two consecutive methods invocations.

The question that may arise now is, what should happen when a contract is broken? Generally, an exception is thrown when a client or a supplier fails to fulfill its obligations. It's important to

```
decrement is
    require
        count > 0
    do
        count := count - 1
    ensure
        count = old count - 1
    end
```

Listing 2.1: An example of Eiffel asserts on a method that decrements a count.

understand that a contract violation should point to a problem that needs to be fixed rather than hidden by exception-handling mechanisms (Meyer, 1997). Therefore, for example, contracts are not intended to validate human input (Aniche, 2022). Still, the development team can decide to disable assertions in runtime or to provide efficient exception handling and use contracts only for documentation purposes.

2.2.3 Programming Languages Support

As previously mentioned, the Eiffel language, conceived by Bertrand Meyer in 1985, is intrinsically associated with contracts. As seen in Listing 2.1, Eiffel provides two keywords - *require* and *ensure* - to specify pre-conditions and post-conditions, respectively. Eiffel also offers a keyword - *invariant* - to specify class invariants (see Listing 2.2). A violation of either assertion will throw an exception indicating a fault. More recently, and first appearing in 2009, Dafny[1] is a programming language based on the concept of "Correct by Construction" that offers native support for specifications in a similar approach to Eiffel.

The first example, Listing 2.1, presents a *decrement* method that deducts the value one from a *count* property. Through the *require* keyword, the method specified the pre-condition that before being called, the client must ensure that the count is greater than zero. Additionally, through the *ensure* keyword, the same method guarantees that after its execution, the counter will be one value less than its previous state. If the client fails to fulfill its obligation — only invoke *decrement* when the *count* is greater than zero — then the method does not guarantee to do its job correctly.

In the second Eiffel example, Listing 2.2, a class invariant is being defined through the *invariant* keyword. In this case, during the object's lifecycle, the *products_count* property needs to be always greater than zero. If this is not verified, the contract is violated, and the program halts its execution.

At one point, C# also offered native support for contracts through the .NET Framework. The *System.Diagnostics.Contracts* namespace provided a way to specify pre-conditions, post-conditions, and object invariants. Although the C# language is out of the scope of this work, it's relevant to note that the .NET team identified that the overwhelming usage of the contracts

[1]Dafny website: `https://dafny.org`

```
class
    SHOPPING_CART
feature
    products_count : INTEGER
invariant
    positive: products_count > 0
end
```

Listing 2.2: An example Eiffel invariant to assert that the products count in a shopping cart is always greater than zero.

feature was for null handling, which is consistent with the existing empirical studies (Estler et al., 2014; Schiller et al., 2014). Therefore, the *Code Contracts* feature was deprecated and replaced by the *nullable references types* (Microsoft Learn contributors, 2021).

More relevant to this present study, Java and Kotlin do not offer native support for contract specifications (Chalin, 2006) unless through the *assert* keyword. Still, some libraries allow Java and Kotlin developers to follow DbC on their projects through some constructs. Later in this document, more specifically in section 3.2, different approaches and constructs that allow developers to specify contracts in Java and Kotlin languages will be presented; still, as a reference to this section, the example on Listing 2.3 shows how by importing the *javax.validation.constraints* package, developers can use some of its annotations to specify pre-conditions and post-conditions. In this example, we use the *@NotNull* annotation to assert that the method *addProductToList* expects a not null input (pre-condition). If this is respected, the method ensures that, through the *@Size* annotation, it will return a list with at least one item (post-condition).

Over the years additional effort has been made to bring a standard contract notation through the Java Modelling Language (JML). Today, this work continues through the OpenJML project (Dietrich et al., 2017).

2.2.4 Advantages and Limitations

The motivation behind DbC is to provide a systematic approach to guarantee software reliability. This is done through implementations that satisfy well-understood specifications known as

```
@Size(min = 1)
List<String> addProductToList(@NotNull String product) {
    ... add product to the list ...
    return list;
}
```

Listing 2.3: An example of using annotations to specify pre-conditions and post-conditions in Java.

contracts (Meyer, 1992). According to Murthy (2018), the degree of reliability (encompassing correctness and robustness as advocated by Meyer (1997)) is correlated to the strength of the contract. Stronger and more expressive contracts have a higher probability of catching errors. The opposite is also true. A program may be correct against a weak contract because this contract does not offer enough expressiveness to verify incorrect behavior (Naumchev, 2019). To overcome this, Wei et al. (2011) propose a tool for automatic inference of strong contracts based on the generally simple ones written by developers.

Another proposal from DbC is to render unnecessary the commonly found method - known as defensive programming - of adding arbitrary, often redundant, checks that increase software complexity (Murthy, 2018). Still, as Aniche (2022) argues, there is a distinction between input validation and contracts, and both practices can be used together.

Theoretical literature has a broad (although not unanimous) consensus that DbC helps to improve software reliability (Murthy, 2018; Wei et al., 2011; Hollunder et al., 2012). The different advantages appointed by various authors assert this consensus. Generally, authors suggest that DbC (i) improves code understanding (Fairbanks, 2019; Naumchev, 2019; Wei et al., 2011; Silva et al., 2020), (ii) helps identify bugs earlier and diagnose the failure (Wei et al., 2011; Aniche, 2022; Casalnuovo et al., 2015; Dietrich et al., 2017; Schiller et al., 2014), and (iii) contributes to better tests (Wei et al., 2011; Aniche, 2022; Schiller et al., 2014; Algarni and Magel, 2018; Tantivongsathaporn and Stearns, 2006).

Firstly, DbC improves code understanding by embedding functional requirements in the code (Naumchev, 2019). Methods can be quickly and better understood by the clear and logical specification of what will happen (post-conditions) and the required state (pre-conditions) that leads to the desired effect. This benefits the developer writing code and the code review process (Fairbanks, 2019). Therefore, as contracts are formally defined for each service, there is a smaller probability of incomprehension between components (client and supplier) that could lead to errors (Silva et al., 2020). However, the same authors state that in some contexts, such as component-based software development, the standard DbC approach is insufficient to specify clear and visible contracts. Since components can be self-contained units of deployment, contracts can be hidden as implementation details and, therefore, not visible to clients.

Secondly, the consequence of a contract violation is the halt of the program execution through an exception throw. In other words, when a pre-condition, post-condition, or class invariant is not satisfied, there is an alert that an unexpected behavior is occurring (Aniche, 2022). Additionally, by gaining insight into whether the exception was thrown by a pre-condition, a post-condition, or a class invariant, we isolate and localize the responsible — whether the client or the supplier — for the faulty behavior.

Lastly, contracts help to write better tests. The goal of a test is to probe that a method implementation matches its specification; therefore, developers can only write effective tests if they understand the system's specifications which can be clearly stated through contracts. This is seen in how pre-conditions allow developers to filter unimportant arguments (Algarni and Magel, 2018). In a study by Tantivongsathaporn and Stearns (2006), the authors claim that adopting DbC also decreases the time needed to perform tests.

Nevertheless, the ability to specify requirements as contracts, held by many as the primary motivation for DbC, is considered to be, by others, a limitation in standard DbC approaches, creating a gap between requirements and code implementation. Naumchev (2019) states that DbC mechanisms fail to capture some formal properties of certain forms, such as abstract data types axioms or temporal properties. The literature also mentions a lack of support for reusing recurrent specifications (Naumchev, 2019; Silva et al., 2020). Those limitations can hinder a more consistent and broad usage of the technique.

Still, authors recognize the advantages of teaching DbC as an integral part of software design and implementation modules. The study conducted by Carvalho et al. (2020) supports the previous statement, where undergraduate students found that learning DbC helped them to understand software concepts better. Additionally, the authors observed that teaching the technique to students without previous programming knowledge was easier than to those with already some skills. Huisman and Monti (2021) designed a visual tool for high school students to learn programming with a focus on contract specifications.

2.2.5 Liskov Substitution Principle and Contracts

The *Liskov Substitution Principle (LSP)* is one of the five SOLID principles. These principles were organized by Robert C. Martin in 2000 to help to build flexible and maintainable object-oriented software. Since then, these principles have been considered part of the “software engineering cannon” (Danisovszky et al., 2019). More specifically, LSP was coined by Barbar Liskov and derived from the concept of Design-by-Contract by Bertrand Meyer.

This principle states that *derived classes should be substitutable for their superclasses*. In other words, if a client class uses a *superclass* and we substitute this use with a *child class*, then it should continue to work correctly (Martin, 2003).

The Listing 2.4 presents an example of a Liskov Substitution Principle violation in Java. It presents a superclass *Account* and a subclass *JuniorAccount*. This subclass inherits the *withdraw* method from the superclass, adding the condition that a junior client can only withdraw a maximum of 100 euros. We now consider a *BankClientService* class that contains a property of *Account* type. According to this principle, the client class is expected to be able to use any of the classes interchangeably, given the superclass specification. This will not be true for this example because while it can use an instance of *Account* to withdraw 200 euros, it will fail to do so if it tries the same with the *JuniorCurrentAccount*; hence, a violation of the LSP.

A violation of this principle is primarily an issue with assumptions. In the previous example, from the designer’s perspective, a *JuniorAccount* is an *Account*. Still, since their behavior is different, it can lead to clients failing when making wrong assumptions. The importance of assumptions is why LSP is fundamental in the context of DbC.

When a superclass is annotated with contracts, the client is informed about the assumptions and the behavior he can rely on when using that class. Therefore, according to LSP, it is expected that each subclass "will expect no more and provide not less" than their superclass (Martin, 2003). In

```java
class Account {
  public int balance;
  void withdraw(int amount) {
    this.balance -= amount;
  }
}

class JuniorAccount extends Account {
  @Override
  void withdraw(int amount) {
    assert(amount <= 100);
    super.withdraw(balance);
  }
}
```

Listing 2.4: An example of a Liskov Substitution Principle violation.

this context, we can reformulate the LSP by stating that a subclass is substitutable for its superclass if and only if:

- If pre-conditions are no stronger than the superclass method.
- If post-conditions are no weaker than the superclass method.

In the previous example of Listing 2.4, the subclass *JuniorAccount* added the pre-condition that *the amount to withdraw must be less or equal to 100* and, therefore, strengthened the pre-condition existing in the superclass that was none; thus, the example violates the Liskov Substitution Principle.

2.3 Empirical Studies

This section explores related work on empirical studies about the usage of contracts by practitioners, including the studies' methodologies and main findings. It also presents important annotation studies and asserts usage while relating their findings to contracts.

2.3.1 On Contracts Usage

Over the years, some empirical studies have investigated how contracts are used in practice and if DbC results effectively in fewer defects. Reviewing these studies, we can conclude that there is little evidence that contracts contribute to increased software reliability. This is contrary to the already mentioned theoretical studies that assert DbC as one of the main techniques for software reliability. Despite this, it can also mean that developers do not use DbC to its potential or that present studies face challenges in adequately measuring quality metrics. Additionally, the different experiments usually agree that most developers don't specify contracts, which means there is low adoption of DbC. For this reason, we may be compelled to agree with Tantivongsathaporn and

Stearns (2006) in that the industry does not widely adopt this technique due to the lack of studies that show a significant positive impact from its adoption.

Blom et al. (2002b) and Tantivongsathaporn and Stearns (2006) conducted studies with group projects composed of university students to understand the correlation of quality and time metrics with the application of DbC. Both works are only able to conclude weak positive indications. They could not confirm any impact on quality (maybe due to the difficulty in measuring quality). Still, they demonstrated that DbC required fewer project person-to-hour resources. Tantivongsathaporn and Stearns (2006) state that DbC contributes to less time spent on writing tests. Following this work, Blom et al. (2002a) presented a DbC-based development strategy and a case study applying it to an enterprise project. In this case, they found evidence that the approach resulted in fewer errors and decreased development time. In another study, after applying DbC to an automobile enterprise project, Zhou et al. (2017) concluded that the technique increased reliability in software components. Components with contracts presented fewer defects than those without contracts.

Chalin (2006) investigated 85 open-source and proprietary Eiffel projects with 7.9 million lines of code (MLOC). This study found that 5% of code lines were asserts. This is a considerable amount, but one needs to remember that the authors investigated Eiffel projects, and contracts are one of the distinctive features of the language. Of these asserts, 50% were pre-conditions, 40% were post-conditions, and 7.1% were class invariants. This preference for pre-conditions is consistent with other studies. Dietrich et al. (2017) reported to have found 22.969 pre-conditions, 112 post-conditions, and 100 invariants in the last version of their dataset programs. Schiller et al. (2014) found that 68% of written contracts were pre-conditions. By intuition, we may say that it is easier and more natural to derive assumptions that must be verified as true for a method to behave correctly (pre-conditions). Additionally, some developers may see unit tests as equivalents to post-conditions since both can be used assert the output of the method (Dietrich et al., 2017). On the other hand, Estler et al. (2014) did not find any preference. Schiller et al. (2014) designed an automatic contract inference tool that was able to infer more post-conditions than pre-conditions. This supports our intuition that post-conditions are less used because they are seen as redundant or because they are more challenging for developers to derive and not because there are fewer post-conditions than pre-conditions.

Schiller et al. (2014) explored 3.5 MLOC from 95 C# projects that use Code Contracts. The authors report 43.823 contract clauses being found. Again, it is relevant to stress that those projects were already known to be using contracts, which doesn't allow us to draw conclusions about the contract's adoption numbers. The authors also categorized contracts into three categories according to their usage: *Common-case contracts* that enforce program properties, *Repetitive contracts* that repeat statements present in the code, and *Application-specific contracts* which enforce richer semantic properties. They found that over 73% of contracts were related to null checking (this is consistent with Estler et al. (2014)). Moreover, the authors explain this lack of expressiveness to annotation burden, lack of support, and lack of training. They also emphasize the importance of creating design patterns alongside tools and libraries.

Estler et al. (2014) analyzed a dataset of 21 Eiffel, C# and Java projects with more than 260 MLOC. As in Schiller et al. (2014), those projects were selected for being known to be equipped with contracts. Therefore, the computed proportion of methods containing specifications (around 40%) may not provide a real insight into how popular is contracts' usage. Those contracts are mostly null checks (for the examined Java projects, more than 88% pre-conditions, more than 28% pre-conditions, and more than 50% invariants include null checks). Contrary to other studies (Schiller et al., 2014; Dietrich et al., 2017) Estler et al. (2014) did not find a developer bias for pre-conditions over post-conditions but found that pre-conditions are typically bigger (with more clauses) than post-conditions. Again, this points to developers being more comfortable with designing pre-conditions. By analyzing the evolution of projects through different versions, the authors concluded that the average number of clauses per specification (contract strength) is stable over time and that the method's implementation changes more frequently than its specification. Still, they warned that, throughout the program's evolution, strengthening contracts might be more frequent than weakening (although they admit that more data is required). This can indicate some unsafe evolution of contracts.

Lastly, Dietrich et al. (2017) investigated 176 of the most popular Java projects in the Maven repository, which translated to over 351 MLOC. The results show that the majority of programs don't use contracts significantly. This is a common perception. Another significant contribution was categorizing six different types of contracts — Conditional Runtime Exceptions (CRE), Assertions, Annotations, and others — and identifying the various contract constructs available in Java or third-party libraries for each category. The authors found that CREs are the most used category, followed by asserts. This is expected since both can be expressed through Java's features without requiring third-party libraries. The authors explain that the dataset can suffer from some bias since it is composed mainly of libraries and not end-user applications, which may explain why the use of annotations is low. They also use that fact to explain the prevalence of pre-conditions over post-conditions. Since libraries are to be used by multiple clients, they use pre-conditions to provide a defensive barrier over their own methods. Still, as mentioned earlier, this dominance of pre-conditions is consistent with other studies (Chalin, 2006; Schiller et al., 2014). The authors also studied contract usage through the program's evolution to find that projects that use contracts maintain their usage stable or even expand it. They also reported some unsafe evolution of contracts. This can happen when a method strengthens its pre-conditions - enforcing new rules that the client may not be prepared to submit to - or weakens its post-conditions - breaking the previous guarantees made to its clients. They also found many violations of the Liskov Substitute Principle (with prevalence in the annotation type). According to this principle, a sub-type can only weaken pre-conditions or strengthen post-conditions and class-invariants from its parent (A.Feldman et al., 2006).

2.3.2 On Asserts Usage

While the Design-by-Contract literature frequently uses the *assertion* word, this should not be confused with the *assert* instruction found in languages like C or Java. Typically, in the context of DbC, *assertion* refers to a way of expressing contract specifications. As previously mentioned, the *assert* keyword may be used to specify contracts, but there are other approaches, such as annotations and CREs, among others. Still, *asserts* are one of the most popular forms of contracts (Dietrich et al., 2017) and, therefore, empirical studies related to this tool are relevant for the discussion on contracts.

Kudrjavets et al. (2006) performed a study on two Microsoft Corporation components with a code base mainly written in C and C++. Their main conclusion is that increased asserts density results in a decrease in faulty density. The authors also reported that, in most cases, equipping files with asserts was more effective for fault detection than some static analysis tools.

Kochhar and Lo (2017) studied a dataset of 185 Apache Java projects available on GitHub to determine the correlation between assertion usage and defects and to underuse how usage relates to code ownership and developer experience. They found that adding asserts contributes to fewer defects, especially when many developers are involved. This agrees with reports from Kudrjavets et al. (2006) but is not supported by Counsell et al. (2017) that analyzed two industrial Java systems and found no evidence that asserts were related to the number of defects. Kochhar and Lo (2017) also concluded that developers with more ownership and experience use asserts more often, which shows that more advanced programmers see it as valuable practice. In line with other previously mentioned studies for contracts (Schiller et al., 2014; Estler et al., 2014), most uses are related to null-checking.

2.3.3 On Annotations Usage

Over the last few years, academia has dedicated significant studies to annotation adoption and usage. Although an annotation doesn't necessarily mean a contract specification, as stated earlier, some annotations can be used as assertions for specifying contracts. Therefore, those studies are valuable when discussing contracts in practice besides showing the community's interest in the empirical investigation of language features. This section discusses important annotation studies and relates their findings to contracts.

There is a general understanding that there is an ever-growing use of annotations among practitioners (Yu et al., 2021; Grazia and Pradel, 2022). Contrary to Dietrich et al. (2017) that found low use of annotations-based contracts, we expect to detect a high usage percentage of this contract category.

Yu et al. (2021) conducted a study on 1,094 GitHub open-source projects to investigate how annotations are used, evolve, and impact error-proneness. They found a median value of 1.707 annotations per project, denoting this language feature's popularity, with some developers overusing it. Most annotations are associated with *String* values (62.57%) and then with *Primitive* types (14.98%). They also found a strong positive correlation between annotations usage and developer ownership and that there is some evidence that their use contributes to fewer faults. A relevant implication made by the authors is the need for better training and tools to help derive better annotations. Other authors made a similar claim for contracts (Schiller et al., 2014). Additionally, developers with higher ownership use annotations more often, which agrees with the findings by Kochhar and Lo (2017) related to assertion usage.

Grazia and Pradel (2022) investigated the evolution of type annotations in 9,655 Python prototypes. This is relevant for contract studying when assuming that some type of annotations can act as contracts. For instance, annotating a method's parameter type as an *Integer* bounds the client to satisfy that requirement and avoids the supplier having to check for the input type in the routine's body. In the same way, annotating a return type gives the client more assurance of what he is expected to receive from the routine. 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. (2021). 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. Estler et al. (2014) reported a similar finding related to the stability of contracts while the program evolves. Also relevant is that most type annotations were associated with parameter and return types and less on variable types. Remembering that not all type annotations may be contracts, this may share some common ground on the developer's bias between pre-conditions, post-conditions, and class invariants. By last, 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.

2.3.4 Summary

The research works presented in the previous sections assert the community's interest in studying language features and their relation with software reliability. However, it is understood that the existing studies are insufficient to provide a comprehensive view of the state of affairs, especially on contract adoption and usage. Most of those works were conducted using a dataset of projects known *a priori* to be using contracts (Schiller et al., 2014; Estler et al., 2014). Other studies are solely focused on the Eiffel language, where the use of contracts is expected (Chalin, 2006). Consequently, there is a large gap in the literature about how popular contracts are in the industry. We consider the works of Dietrich et al. (2017) an important milestone in contract empirical research. Still, that work examined mostly library applications which can lead to a bias in the results. This is mostly seen in how the annotations studies showed a rise in its popularity and even an overuse, but (Dietrich et al., 2017) found low usage of this feature. Therefore, the research opportunities on contracts are still many and of interest to the community.

Chapter 3

Methodology

This chapter presents and describes the proposed approach to address the research questions proposed in chapter 1. Firstly, section 3.1 describes the dataset, including its origin and building process. Then, section 3.2 focus on a central part of this work: listing the constructs and defining the contracts categories we propose to study in this work. Finally, section 3.3 explains how the contract analysis tool examines the dataset to produce the data required to answer each research question focusing on its three main components: the *usage*, *evolution*, and *inheritance* studies.

3.1 Dataset

This work examines how and to what extent contracts are used in Android applications. As more applications are explored, we can assert with a higher certainty that the results propose a more complete and reliable representation of reality. Therefore, the value of this study broadly correlates to the number of applications analyzed.

In this section, we first introduce the source of the Android projects, which will be evaluated through the analysis tool. Then, we outline the pipeline that creates and prepares the dataset containing those projects.

3.1.1 Data Source

As this work aims to study the presence and use of contracts by practitioners, our dataset is composed of real-world applications. These applications were obtained from F-droid[1], 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 (with public source code) on a wide range of domains makes F-Droid a good option. Apart from native Android applications written in Java or Kotlin, the catalog also contains projects that use hybrid frameworks (e.g., React Native) that we must exclude from our dataset.

[1] `https://f-droid.org` (accessed 6 June 2023)

3.1.2 Dataset Construction

The dataset that will be evaluated by the contracts tool is built with a pipeline consisting of various Python scripts. This pipeline filters, downloads, and structures the dataset to be easily read by the analysis tool. Figure 3.1 presents all steps in the pipeline.

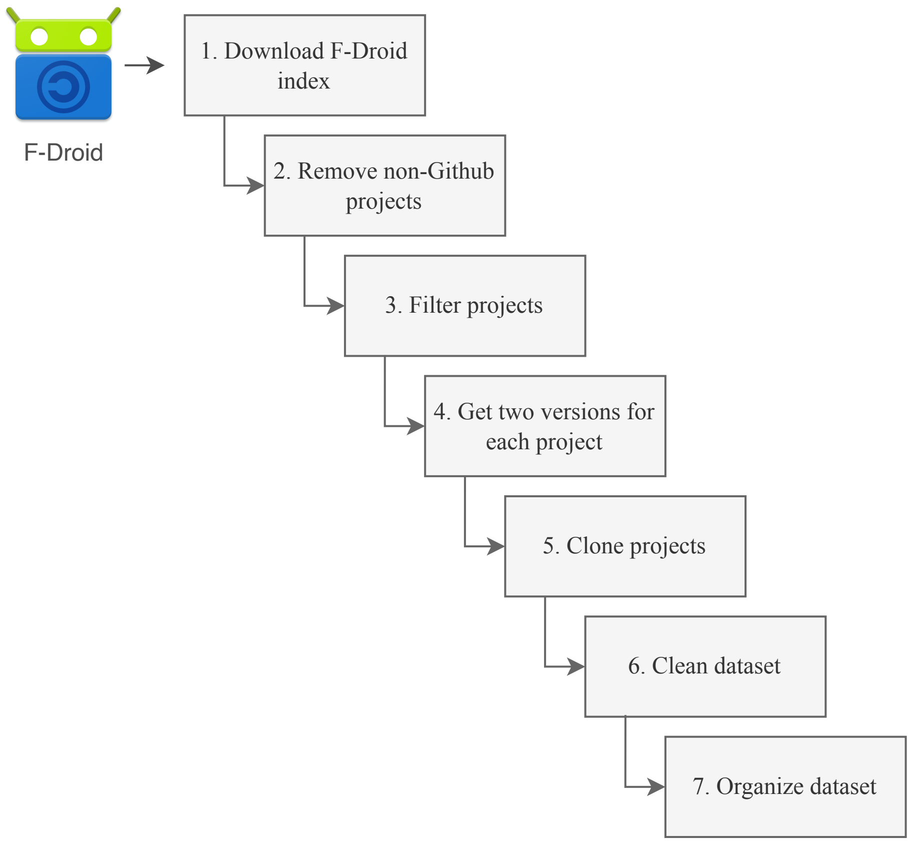

F-Droid
1. Download F-Droid index
2. Remove non-Github projects
3. Filter projects
4. Get two versions for each project
5. Clone projects
6. Clean dataset
7. Organize dataset

Figure 3.1: Dataset pipeline

The first step in the pipeline is to *download the F-Droid index*, which is a list of URLs for each project available in the catalog. Excluding this step, all the following steps are data source independent, meaning the pipeline can easily be used with origins other than F-Droid.

Next, we *filter* projects listed in this F-Droid index based on the following criteria:

- The application source code is hosted in GitHub;
- The application source code is either Java or Kotlin;
- The GitHub project is not archived;
- The GitHub project has had a commit since 2018.

These inclusion criteria ensure that the project's source code is easily accessible (through GitHub, written mainly in Java or Kotlin (the languages we are interested in studying) while guaranteeing the project is active and relevant. This step also retrieves additional meta-information about each project, including 1) the application name; 2) the GitHub project URL; 3) the source code primary language; 4) the number of GitHub watchers; 5) the number of GitHub stars; 6) the number of contributors; 7) the last commit date; 8) the number of merged pull requests; 9) the number of closed pull requests; and 10) the percentage of accepted pull requests. This information can help to characterize the dataset.

We try to *retrieve two versions* for each one of the filtered projects, which is a required step for the evolution study. We do this by storing a list of the URLs pointing to the two GitHub release versions. 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.

Finally, the pipeline *clones the projects* contained in the versions list. As an optional step, *every file that is neither a Java nor a Kotlin file is removed* from the dataset, which helps to decrease its size. In the end, the project folders are zipped and organized in the structure expected by the contracts tool, as exemplified next:

```
dataset
├── accountName-projectName
│   ├── accountName-projectName-0.zip
│   └── accountName-projectName-1.zip
└── ...
```

All program versions are grouped in a main folder labeled with the GitHub account and the project's names. Each version contains as a suffix its index (*0* or *1*) according to whether they are the first version or the last. This is the dataset folder that the analysis tool will use to examine the contracts' presence and usage.

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 inclusion criteria. This dataset translated to an overall size of 138,45GB. However, after applying the optimization script to delete non-desirable files, like images, the size was reduced to 98,26GB.

3.2 Contracts Constructs and Classification

As introduced in section subsection 2.2.3, Java and Kotlin do not offer a native approach to declare contracts as seen in languages like Eiffel. Still, there are language features and libraries — e.g. annotations, exceptions, and asserts — that can be used to express contracts in Java and Kotlin programs. Therefore, it's crucial to identify those constructs and categorize them. The contracts tool needs to be able to capture those constructs in the analyzed source code.

We adopted and supported the five categories proposed by Dietrich et al. (2017), which include Conditional Runtime Exceptions (CREs), APIs, Assertions, Annotations, and Others.

3.2.1 Conditional Runtime Exceptions

When an unhandled exception is thrown, the program halts its execution. This means that an exception can be used to signal a contract violation. It's important to note that the exception itself does not represent a contract but needs to be associated with a previous check — for example, an exception thrown inside an *if-else block* — to be considered so.

The Java (and Kotlin) language offers 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*. Appendix A lists all exceptions being considered by our analysis tool. We expanded the list proposed by Dietrich et al. (2017) from 6 to 74 exceptions. Since the exception does not restrict any use case, it depends on the practitioner's interpretation, and the context, if the exception is associated with a contract specification or not. We are also 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. (2017) argues, this is the strongest possible pre-condition and can not be satisfied by any client.

In the example in Listing 3.1 we identify the presence of an *IllegalArgumentException* that is thrown when the contract *shoppingCart.isEmpty()* is not respected. In this case, the method *proceedWithCheckout* states that it can only perform its task when the *shoppingCart* has at least one item. Therefore, this is an example of a pre-condition.

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

Listing 3.1: An example of a method that uses an IllegalArgumentException to signal a contract violation.

3.2.2 APIs

Over the years, some libraries have proposed APIs that consist of wrappers around conditional exceptions and other basic constructs. This contributes to a simpler, more verbose, and explicit representation of contracts. We are interested in the four APIs listed in Table 3.1 and, more particularly, in the methods presented in Appendix B.

Table 3.1: List of APIs that provide contract representations.

APIs packages
org.apache.commons.lang.Validate.*
org.apache.commons.lang3.Validate.*
com.google.common.base.Preconditions.*
org.springframework.util.Assert.*

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

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

Listing 3.2: An example of a Kotlin method that uses the notEmpty method from the Apache's Validate class to specify a pre-condition.

In the first case, *Apache Commons* offers the *Validate*[2] class that according to the official documentation "assists in validating arguments", which suggests a pre-condition usage. The same documentation also states that the validation methods follow the following principles:

- A null argument leads to a *NullPointerException*.
- A non-null argument leads to an *IllegalArgumentException*.
- An index issue in a collection-type structure leads to an *IndexOutOfBoundsException*.

We can understand from those principles that the methods provided by the *Validate* class are simply wrapping exceptions that we have already listed in section 3.2.1. Still, we can see in the example of Listing 3.2 that this API contributes to cleaner code compared to a raw CRE-based solution since we can specify the contracts in a single line and with meaningful wording. In this example, we are declaring a pre-condition *items list is not empty*. 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.

Another example of an API is the *Guava* library from Google, which includes the *Preconditions*[3] class. In this case, the official documentation clearly states that the class includes "static convenience methods that help a method or constructor check whether it was invoked correctly (whether its pre-conditions have been met)". In the example of Listing 3.3 we specify the pre-condition that the *membership discount can only be applied when the user has the member status*.

Lastly, *Spring Framework* also provides the *Assert*[4] class with methods to assist in validating arguments. In the example of Listing 3.4 we specify the pre-condition that the order's items must not be empty. If this happens, the assert throws an *IllegalArgumentException*. Since this library is to be used within the *Spring* framework, we are not expected to find occurrences of its use in the

[2]https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html (accessed 4 June 2023)

[3]https://guava.dev/releases/19.0/api/docs/com/google/common/base/Preconditions.html (accessed 4 June 2023)

[4]https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/Assert.html (accessed 4 June 2023)

```
import com.google.common.base.Preconditions

fun applyMembershipDiscount(user: User) {
    Preconditions.checkArgument(user.isMember)
    ...
}
```

Listing 3.3: An example of a Kotlin method that uses the checkArgument method from the Guavas's Preconditions class to specify a pre-condition.

Android applications we'll analyze. However, as we want to continue to support what (Dietrich et al., 2017) presented in their study, we are still including it in ours too.

As we can perceive from the three different APIs and their documentation, they are built to be used as pre-conditions. The same libraries do not offer any equal approach to specify post-conditions which clearly shows a bias from tool builders into pre-conditions over post-conditions. Nevertheless, and against the documentation guidelines, practitioners can still use any of those API's methods to check post-conditions.

3.2.3 Assertions

Assertions have been introduced in Java 1.4 and are specified through the *assert* reserved keyword. It helps practitioners to 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 the practitioner may be assuming that the 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 easily be used to check pre and post-conditions as shown in the example in Listing 3.5. In this case, the contract associated with the *addToShoppingCart* method defines two pre-conditions - 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 post-condition - the items added to the shopping cart *will be present in the shopping cart list*.

```
import org.springframework.util.Assert

fun createOrder(order: Order) {
    Assert.notEmpty(order.items, "Order must have at least one item")
    ...
}
```

Listing 3.4: An example of a Kotlin method that uses the notEmpty method from the Spring's Assert class to specify a pre-condition.

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

Listing 3.5: An example of a Java method that uses the asserts to specify pre and post-conditions.

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 the analysis tool to distinguish between Kotlin's assert or a developer's custom method that does something else. Additionally, contrary to Java, Kotlin always executes the assert expression and only uses the *-ea* JVM flag to decide whether to throw the exception, which can cause problems on performance-sensitive applications.

Additionally, Kotlin offers other methods — *check()*, *checkNotNull()*, *require()*, and *requireNotNull()* — which, although they throw an *IllegalArgumentException* or an *IllegalStateException* instead of an *AssertionError*, they were added to the assertions category because of their syntactic similarities. The example on Listing 3.6 uses Kotlin's methods to specify the same pre and post-conditions as in the previous Java example on Listing 3.5.

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

Listing 3.6: An example of a Kotlin method that uses the assert, require and check methods to specify pre and post-conditions.

3.2.4 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 3.2 lists the annotation packages we are particularly interested in studying in the context of contracts. This is an expansion over Dietrich et al. (2017)'s original list with the addition of *android.annotation* and the *androidx.annotation* packages. Appendix C provides a more the list of annotations analyzed from each package.

Table 3.2: List of packages contains annotations for contracts specification.

Annotation packages
javax.annotation.* (JSR305)
javax.annotation.concurrent.* (JSR305)
javax.validation.constraints.* (JSR303, JSR349)
org.jetbrains.annotations.*
org.intellij.lang.annotations.*
edu.umd.cs.findbugs.annotations.*
android.annotation.*
androidx.annotation.*

The annotation-based approach is particularly interesting due to two reasons. Firstly, many annotations can be associated with the method's arguments (pre-conditions), the method's return values (post-conditions), or the class properties (invariants). Secondly, 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, which happens with CREs, APIs, and assertion-based approaches.

In the example in Listing 3.7 we see how we can use annotations from the *javax.validation-.constraints.** packages to specify contracts. The method states that it can only perform its job, assuring that it will *return a list with a minimum size of 1* (post-condition), if the *item identifier is not null* and the *quantity is greater or equal to one* (pre-conditions). At the same time, the class property *items* is associated with a class invariant that states that the *shopping cart can only contain ten items at maximum*. Revisiting our earlier statement, we can see in this example that adding contracts through annotations does not require adding extra checks to the method's 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> {
        ...
    }
}
```

Listing 3.7: An example of a Kotlin class that uses annotations from javax.validation.constraints.* to specify contracts.

```
fun sendBirthdayMessage(birthdate: String?) {
    isBirthdateValidOrElseThrow(birthdate)
    val birthdaySplit = birthdate.split("/")  // compilation error
    ...
}

fun isBirthdateValidOrElseThrow(birthdate: String?) {
    if (birthdate == null) {
        throw IllegalArgumentException()
    }
    ...
}
```

Listing 3.8: An example of Kotlin code that does not compile due to a non-safe call on the split method.

3.2.5 Others

As mentioned in chapter 2, there are open initiatives in the Java world, like OpenJML and jContractor, to bring a standard contract notation. As Dietrich et al. (2017) argues, their use is rare and, therefore, we are also not including them in our study.

Still, we extended the analysis tool to support a particular type of contract introduced in Kotlin 1.3. Kotlin Contracts are an experimental feature that allows the developer to state a method's behavior to the compiler explicitly.

Let's consider the example on Listing 3.8. There is a method *sendBirthdayMessage* that splits the *birthdate* string by the '/' character. Before trying to split the string, a second method is invoked to perform some validations in the *birthdate*, including checking whether it is null. Although we can understand that when the control reaches the *split* call, the *birthdate* will never be null due to the validation in the *isBirthDateValidOrElseThrow* method, the compiler is not able to understand so, throwing the error *"Kotlin: Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?"*.

This is the kind of situation — when the developer knows more than the compiler — in that *Kotlin contracts* can be useful to establish a contract between the two parts. In the example of Listing 3.9 we see how to use the *contract* expression to assure the compiler that if the method returns, then the *birthdate* will not be null. In this example, the compiler knows that *split* is secure since *birthdate* will never be null at that point.

This feature, although still liable to receive changes in the future, offers more capabilities that broaden its use but that are out of this document's scope. Since it is based on pre-conditions and post-conditions to specify a contract between two parts — the developer and the compiler — we decided to include it in our study.

```kotlin
import kotlin.contracts.ExperimentalContracts
import kotlin.contracts.contract

@ExperimentalContracts
fun sendBirthdayMessage(birthdate: String?) {
    isBirthdateValidOrElseThrow(birthdate)
    val birthdaySplit = birthdate.split("/")  // compiles without error
    ...
}

@ExperimentalContracts
private fun isBirthdateValid(birthdate: String?) {
    contract { returns() implies (birthdate != null) }
    if (birthdate == null) {
        throw IllegalArgumentException()
    }
    ...
}
```

Listing 3.9: An example of Kotlin code that uses Kotlin Contracts to improve smart-cast analysis.

3.3 Analysis Tool

After stating in section 3.2 the constructs we are interested in identifying, this section describes the functioning of the analysis tool that evaluates the dataset and produces the required data to support this work's proposed research questions.

3.3.1 Overview

The proposed analysis tool, which is at the core of this work, is based on the one created by Dietrich et al. (2017) to examine the presence and usage of contracts in Java applications. Our major contribution to the tool was extending it to support Kotlin source code and more constructs focused on Android applications. Additionally, the framework's code also suffered considerable refactoring and organization to simplify and ease its comprehension and maintainability.

The main effort was to add support for Kotlin source code. The original tool was using the JavaParser[5] 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[6] to perform this task. This required us to implement new versions of the tool's extractors and visitors classes using the methods 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.

A high-level representation of this tool can be seen in Figure 3.2, where each block corresponds to a script written in Java. Although it is possible to change the order in which these scripts

[5]https://javaparser.org (accessed 4 June 2023)
[6]https://github.com/JetBrains/kotlin (accessed 4 June 2023)

are run, there are some dependencies between them since some scripts may require data produced by previous ones.

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

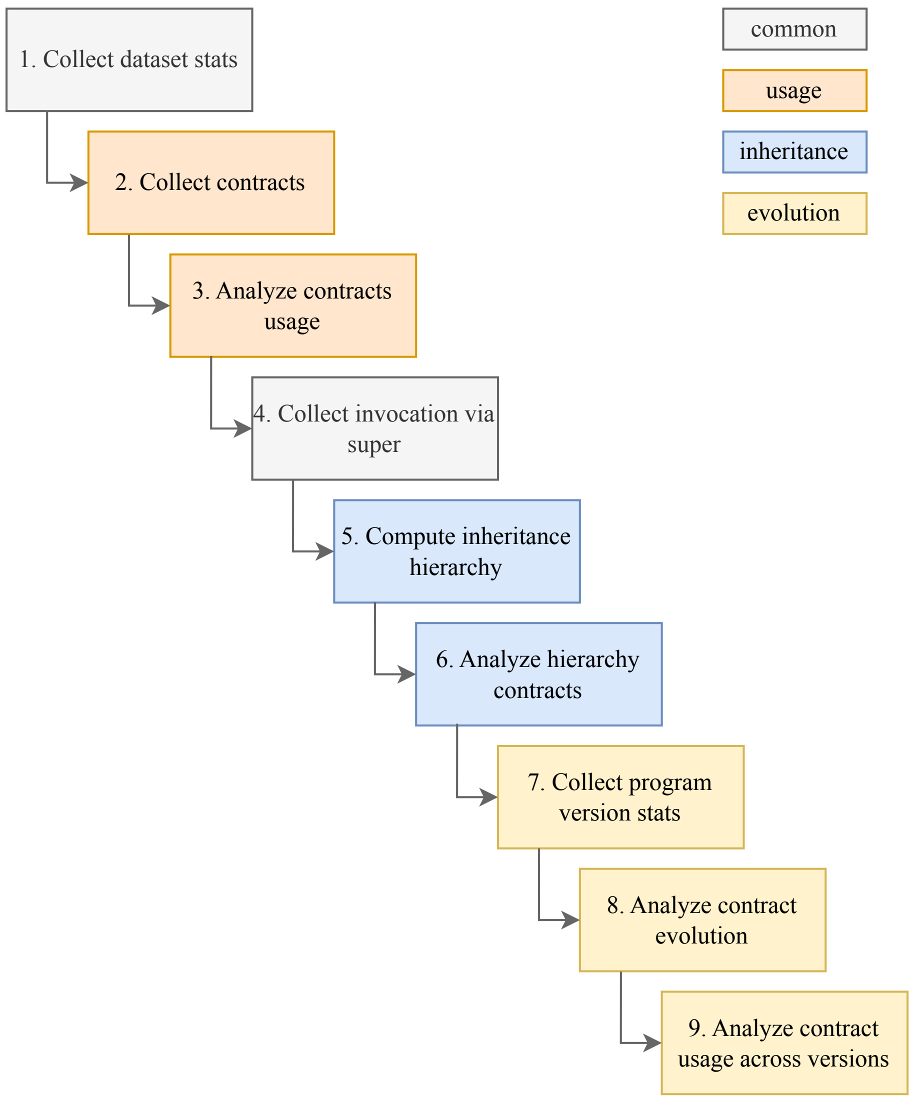

1. Collect dataset stats
2. Collect contracts
3. Analyze contracts usage
4. Collect invocation via super
5. Compute inheritance hierarchy
6. Analyze hierarchy contracts
7. Collect program version stats
8. Analyze contract evolution
9. Analyze contract usage across versions
common
usage
inheritance
evolution

Figure 3.2: An high-level overview of the analysis tool structure.

3.3.2 Usage Study

The usage study occurs in two main steps: identifying contract occurrences and producing statistics about those results.

The proposed tool uses the JavaParser and JetBrains's Kotlin compiler libraries to perform AST analysis of all dataset's source code file that is either Java or Kotlin. 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 Appendix A). In line with Java's good practices, we assume that CREs are used with pre-conditions.

- *APIs*. Firstly, we check whether the file contains an import declaration to any API package listed in Table 3.1. 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 in subsection 3.2.2, we assume the analyzed APIs to be associated with pre-conditions.

- *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 expressions — *assert*, *require*, *requireNotNull*, *check*, *checkNotNull*. Next, we identify whether the class invokes any method with one of those names. Suppose a class has a method declaration/import statement and an invocation with one of these expression's names. In that case, we consider it an ambiguous situation, and therefore, we don't consider it an assert instance. If the class invokes one of those methods but doesn't declare/import any method with that same name, we consider it an assert. This is not a fool-proof approach, but it minimizes the under-reporting to a residual level. We do not classify assertions either as pre-conditions or post-conditions.

- *Annotations*. We check if the source code file contains an import statement to one of the packages listed in Table 3.2. If that's 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: 1) annotations associated with a method's parameters are pre-conditions; 2) annotations associated with a method are post-conditions; 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>)}*.

In the end, a *JSON* file is created for each program-version pair to store the identified contracts, including 1) the file path, 2) the associated condition, 2) the method or property name, 3) the type of artifact (method or property), 4) the line number, and 5) the contract type. As we'll see in the following sections, this information is crucial for the next steps of the experimental studies.

```
[
    {
        "cu": "app/src/main/java/health-tracker/RippleDrawable.java",
        "condition": "maxRadius != RADIUS_AUTO && maxRadius < 0",
        "method": "setMaxRadius(int)",
        "additional_info": "maxRadius must be RADIUS_AUTO or >= 0",
        "artefact_type": "METHOD",
        "line": 607,
        "name": "health-tracker",
        "abstract_method": false,
        "type": "CREIllegalArgumentException",
        "version": "0"
    }
]
```

Listing 3.10: An example of a JSON file that stores the identified contracts in a program-version pair.

The Listing 3.10 shows an example of this *JSON* file for the version *"0"* (first version) of a program named *"health-tracker"*. In this example, an *"IllegalArgumentException"* contract was found in the *"non-abstract"* method *"setMaxRadius(int)"* in the line *"60"* of the class *"RippleDrawable"*. This exception is thrown when the condition *"maxRadius != RADIUS_AUTO && maxRadius < 0"* fails and is associated with the message *"maxRadius must be RADIUS_AUTO or >= 0"*.

In the second step of the *usage* study, the *JSON* files are analyzed to produce statistics about the identified contracts, including the frequency of each category (API, annotation, assertion, ...), class (pre-conditions, post-conditions, and class invariants), construct (java assert, Guava API, *androidx* annotations, ...), to compute the Gini coefficient for each category, and to list the programs with more contracts for each category.

3.3.3 Evolution Study

In the evolution study, we are interested in knowing what happens to a contract while the application evolves. In other words, after identifying a contract in the first version of the application, we'll try to see if, in the later version, the contract still exists, was modified or removed. At the same time, we are also reporting cases when a contract was added to an artifact (method or parameter) in the later version of the app that was not present in the first. This information provides insights into how contracts evolve in an application and whether this evolution proposes risks to the client.

At this point, we reiterate that a contract establishes rights and obligations for each part — the client assumes that the supplier will keep its obligations and vice-versa. Therefore, when a contract is altered, both parts should be informed and updated accordingly. This is particularly crucial when a *pre-condition is strengthened* or when a *post-condition is weakened*. In the first case, if the pre-condition is strengthened and the client does not know it, it can fail to cover its

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

Listing 3.11: A Java method signature in version 0 of an application specifying three pre-conditions.

new obligations, and, therefore, the supplier is not bound to keep its part of the contract. In the latter case, if the post-condition is weakened, the client may still be making assumptions that the supplier does not ensure anymore.

To better illustrate this, let us consider the examples on Listing 3.11 and on Listing 3.12. In the first example, we have a method signature in the first version of the application, and in the second, we have the same method but in the last version. We identify that the annotation @*NonNull* was added to the *toolbar* parameter in the last version. This is the case of a *pre-condition 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.

To conduct this study, we follow the same approach as Dietrich et al. (2017), firstly creating *diff records* from the contracts present at the two versions of a program's method and then classifying them according to the *evolution patterns* listed in Table 3.3.

The algorithm to create *diff records* is illustrated in Figure 3.3. This is performed by walking through each contract identified in the *usage* study (steps 1 and 2). We create a unique index for each contract in the loop to ensure we are not double-counting occurrences (steps 3 and 4). If the contract was not analyzed yet, we determine whether the contract belongs to the first version of the application (step 5). If this is the case, we create a *diff record* by retrieving all the contracts in both versions of this contract's method (steps 5.a and 5.b). Otherwise, if the contract belongs to the last version of the application (step 6), we determine whether the associated method already existed

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

Listing 3.12: A Java method signature in version 1 of an application specifying four pre-conditions.

in the first version (step 7). If the method existed and its first version didn't contain contracts, we create a diff record with only the last version's contracts (steps 8 and 9). If the first version contains contracts, we don't create a *diff records* to not double-count contracts since they will be reported by step 5.b. Ultimately, the program outputs the diff records created for each program-version method (step 10).

After creating the *diff records* that compare the contracts found in the first and last version of a program's method, the analysis tool classifies each record according to the evolution patterns listed in Table 3.3. Although some changes are impossible to classify automatically without manual input, most simple cases are successfully classified. The analysis tool also reports instances where the constraints are the same — the specification was stable — or when a *diff record* could not be classified.

Table 3.3: Classification of the diff records produced during the evolution and LSP study.

Classification	Description	Risk
PreconditionsStrengthened	A pre-condition was added to a method or a clause to an existing pre-condition with the '&' or '&&' operators.	Potential risk
PreconditionsWeakened	A pre-condition was removed from a method, or a clause was added to an existing pre-condition with the '|' or '||' operators.	No risk.
PostconditionsStrengthened	A post-condition was added to a method or a clause to an existing post-condition with the '&' or '&&' operators.	No risk.
PostconditionsWeakened	A post-condition was removed from a method, or a clause was added to an existing post-condition with the '|' or '||' operators.	Potential risk.

3.3.4 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 and as explained in subsection 2.2.5, proper handling of contracts should follow the Liskov Substitution Principle, 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 pre-conditions* and *strengthen post-conditions*.

This study identifies the potential misuse of contracts in an inheritance context, namely instances of violations of the Liskov Substitution Principle. An example of this situation is shown in Listing 3.14 and Listing 3.15. In the first code example, we have the *TagEntry* class that extends the *EntryItem* class. It also overrides the *setName* method inherited from its parent class. In the second code example, we see the *setName* method implementation in the superclass *EntryItem*.

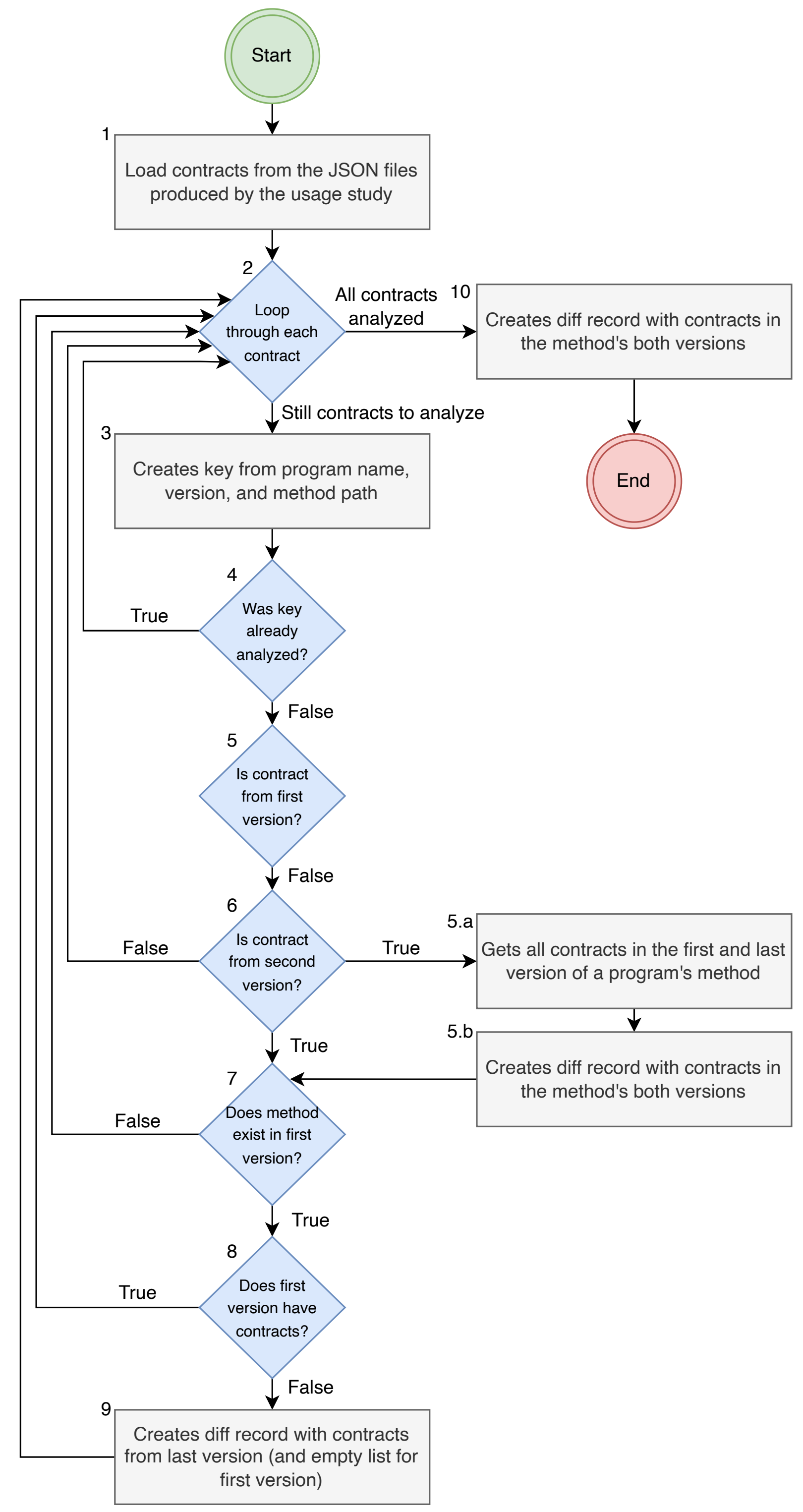

Start
1
Load contracts from the JSON files produced by the usage study
2
Loop through each contract
All contracts analyzed
10
Creates diff record with contracts in the method's both versions
End
Still contracts to analyze
3
Creates key from program name, version, and method path
4
Was key already analyzed?
True
False
5
Is contract from first version?
False
6
Is contract from second version?
False
True
5.a
Gets all contracts in the first and last version of a program's method
5.b
Creates diff record with contracts in the method's both versions
True
7
Does method exist in first version?
False
True
8
Does first version have contracts?
True
False
9
Creates diff record with contracts from last version (and empty list for first version)

Figure 3.3: An overview of the algorithm to create *diff records* of the contracts found in two versions of a method.

```
[
    {
        "methods": [
          "getUserWithUsernameAndServer(String,String)",
          "deleteUser(User)",
          "updateUser(User)",
          "getUser()",
        ],
        "cuName": "app/src/main/java/com/health-tracker/user/
            UsersRepositoryImpl.kt",
        "className": "user.UsersRepositoryImpl",
        "parents": [
          {
            "programName": "health-tracker",
            "cuName": "app/src/main/java/com/health-tracker/user/
                UsersRepository.kt",
            "className": "user.UsersRepository",
            "programVersion": "1"
          }
        ]
      }
]
```

Listing 3.13: An example of a JSON file that stores a class's methods and parents.

While the superclass implementation contains no contract, it was added to the subclass implementation a CRE pre-condition throwing an *IllegalStateException* when the *id* property does not end with the *name* parameter. Therefore, we are in the presence of a *pre-condition strengthening* in the context of *inheritance*, i.e., a violation of the Liskov Substitution Principle.

To detect those occurrences, the analysis tool first lists all methods in each program-version pair associated with their respective class. Additionally, it also identifies the class' parents. Listing 3.13 shows the example of *JSON* file that stores these information for the program *"Health Tracker"* in version *"1"* (last version). In this case, we observe that the *"UsersRepositoryImpl"* class contains *"four"* methods and implements the interface *"UserRepository"* (parent).

Then, the *LSP* study follows a similar approach to the *evolution* study and its *diff records* creation algorithm presented in Figure 3.3. The difference is that the *diff records* are now created between the subclass and the superclass methods. These records are classified according to the evolution patterns described in Table 3.3.

```java
public class TagEntry extends EntryItem {
    public final String id;

    @Override
    public void setName(String name) {
        if (name != null) {
            if (!id.endsWith(name))
                throw new IllegalStateException("The display name and tag name
                    need to be equal.");
            super.setName(name);
        } else {
            super.setName(id.substring(SCHEME.length()));
        }
    }
}
```

Listing 3.14: A Java class that overrides the setName method from its parent class.

```java
public class EntryItem {
    public void setName(String name) {
        if (name != null) {
            this.name = name;
            this.normalizedName = StringNormalizer.normalizeWithResult(this.
                name, false);
        } else {
            this.name = "null";
            this.normalizedName = null;
        }
    }
}
```

Listing 3.15: A Java class that provides a setName method.

Chapter 4

Results

After having described the methodology in the last chapter, we now present the results of the large-scale empirical study on contracts using an Android dataset.

First, section 4.1 presents metrics about the evaluated dataset, including its dimensions. The section 4.2 provides insights into how and to what extent contracts are used in Android applications (RQ1), then section 4.3 answers whether the number of contracts increases proportional to the program's size (RQ2), and, finally, section 4.4 gives a perspective on the practitioner's safety while using contracts in program evolution and inheritance (RQ3).

4.1 Dataset

As described in subsection 3.1.2, the dataset was built from Android applications listed in the F-Droid catalog following a process of filtering described in the same section. The Table 4.1 presents metrics about the final dataset size used in the empirical evaluation. After applying the inclusion criteria, we ended up with *2,390 different applications* distributed in *1,767 Java* and in *623 Kotlin* applications. Since we tried to retrieve two versions for each application, we analyzed *4,192 program-version pairs*. This means that for 294 applications, it was only possible to retrieve a single version. While these applications are still evaluated in the context of the *usage* and *LSP* studies, they are not considered for the *evolution* study. Nevertheless, these numbers support our proposal to conduct a large-scale empirical study on contracts using a much bigger dataset compared to all the contract studies discussed in section 2.3.

Additionally, from the same table, we observe that the dataset is biased toward Java applications. The dataset includes 204,478 Java and 123,222 Kotlin compilation units and, therefore, Java represents 62.4% of the overall number of compilation units. This bias 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 551,456 classes, 2,682,160 methods, and 300,639 constructors. As we didn't consider *private* methods because those methods are not used directly by a client, and a contract is a bond between a supplier and a client, we analyzed 2,532,399 *public*, *protected*, and *internal* methods and constructors.

Table 4.1: Dataset metrics.

metric	Java	Kotlin	Both
compilation units	204,468	123,222	327,690
classes	299,824	251,632	551,456
methods (all)	2,079,276	602,884	2,682,160
constructors (all)	205,773	94,866	300,639
methods (public, protected, internal)	1,770,972	482,972	2,253,944
constructors (public, protected, internal)	184,868	93,587	278,455
KLOC including comments	40,041	11,708	51,749

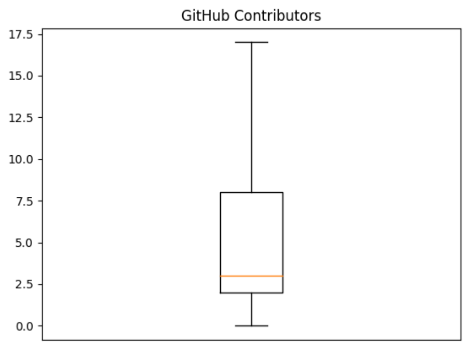
From the standpoint of the dataset's diversity, it includes apps from various domains, such as gaming, communication, multimedia, security, health, and productivity, among others. Likewise, there is also a desirable heterogeneity in the project's characteristics that contribute to the richness of this study and assert its findings as a fitting representation of reality. Figure 4.1 shows the distribution of GitHub-related metrics — including the number of *contributors*, *stars*, *watchers*, and *forks* — for the projects that form the evaluated dataset. While the number of *contributors* describes the project's team and its size, the number of *stars*, *watchers*, and *forks* help to assess each project's popularity and relevance among other developers. For reference, the maximum outlier for each metric is 1682 for watchers, 33,689 for stars, 11,633 for forks, and 398 for contributors. Again, this diversity ensures the quality of the dataset and reduces potential bias.

It is also worth noting the difference in the number of *compilation units* and *classes* between the two languages. Although the dataset contains 1.67 times more Java compilation units than Kotlin ones, it only includes 1.19 times more Java classes. Since a compilation unit usually represents a file, this means that there are more classes per file in Kotlin (2.04) than in Java (1.47). This is expected due to the more restricted Java rules that, for example, only allow a single top-level public class per file.

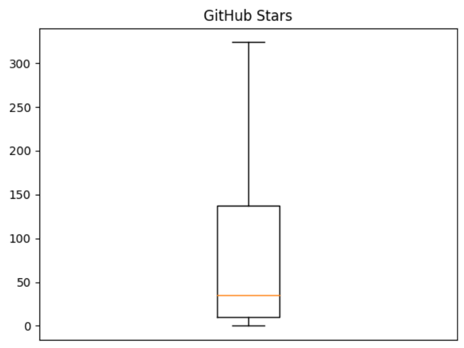
4.2 RQ1: How and to what extent are contracts used in Android applications?

The first research question is concerned with analyzing how popular contracts are among Android application developers. Additionally, it also proposes to understand some of the practitioners' habits and preferences in the context of contract usage. The findings presented in this section are supported by the results produced by the *usage* study detailed in subsection 3.3.2.

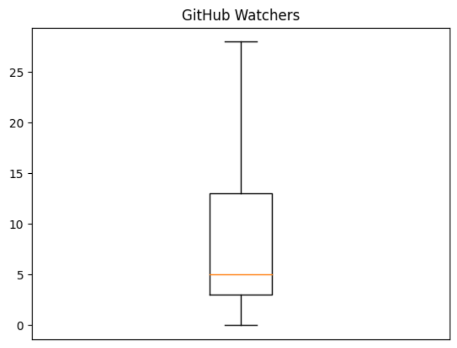
Table 4.2 shows the number of contracts found per category, considering all versions (columns 2 and 3) and considering only the latest version of each application (columns 4 and 5). The same table also identifies the number of applications containing at least one contract for that category (columns 6 and 7). From the table's data, the most obvious conclusion is that, in both languages,



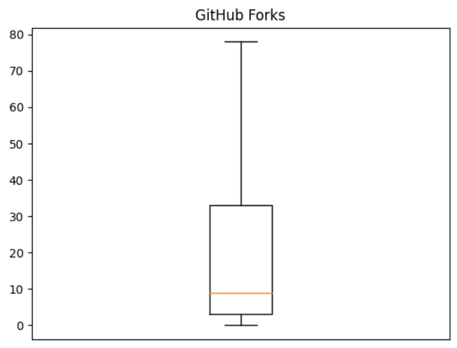
(a) Number of contributors.



(b) Number of stars.



(c) Number of watchers.



(d) Number of forks.

Figure 4.1: The distribution of GitHub repositories-related metrics for the dataset's projects, without outliers.

annotation-based contracts are the most popular category. More specifically, considering both languages in the last version, annotations represent 87.1% of the contracts found, followed by CRE with 9.7%, and then assertions with 2.5%. 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 by our inclusion of the four language's standard library methods listed in subsection 3.2.3, where *require()* alone counts 901 total occurrences distributed in 112 last versions.

Finding 1: Most contracts are annotation-based, accounting for 88.31% in Java and 77.44% in Kotlin of the total number of contracts found.

This distribution in categories' popularity significantly differs from the findings of Dietrich et al. (2017). The authors reported that the most common category was CREs and found surprisingly low use of annotations. This may be explained by the difference between the dataset's nature. While our dataset is formed mostly of user-focused Android applications, the author's dataset was

Table 4.2: 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	1,121	9	24	4
annotation	158,400	24,125	139,507	15,068	1,097	541
assertion	3,525	3,746	2,186	2,239	326	232
CRE	26,061	3,292	15,150	2,139	789	287
other	-	1	-	1	-	1

mainly Java libraries. In Table 4.3, we can also see that most annotations found to belong to the *androidx.annotation.** package that the authors didn't 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 (Yu et al., 2021; Grazia and Pradel, 2022).

From Table 4.2, we also verify that the usage of *APIs* is very low in both languages, and it is even more residual in Kotlin applications, where only nine instances were found in the latest versions. The known industry skepticism around adding third-party dependencies to projects, which may lead to maintainability and support issues in the future, may explain this finding (Backes et al., 2016; Wang et al., 2020).

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

In Table 4.3, we have a more detailed perspective by having the frequency of each construct. Firstly, we again 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 4.4, which presents each category's computed *Gini coefficient*. The *Gini coefficient* measures the inequality among the values of a frequency distribution. In other words, a *Gini coefficient* of 0 indicates perfect equality, where all applications 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 higher than 0.50, except for Kotlin's API usage. The fact that almost all coefficients are very high (close to 1) means that although some applications use contracts intensively, the majority do not use them significantly. This aligns with the results found by Dietrich et al. (2017). This conclusion can also be seen from Table 4.5 that lists the five projects that use more contracts per category. We find that a small group of projects

Table 4.3: Number of contracts found in the dataset by construct and category.

		contracts (all ver.)		contracts (2nd ver.)		applications	
Construct	Category	Java	Kotlin	Java	Kotlin	Java	Kotlin
cond. runtime exc.	CRE	25,565	3,232	14,887	2,071	779	285
unsupp. op. exc.	CRE	511	142	308	116	97	27
java assert	assertion	3,525	-	2,217	-	325	-
kotlin assert	assertion	-	3,868	-	2,370	-	234
guava precond.	API	1,798	10	1,121	9	22	4
commons validate	API	148	0	3	0	1	0
spring assert	API	1	0	1	0	1	0
JSR303, JSR349	annotation	0	0	0	0	0	0
JSR305	annotation	4,195	20	2,133	13	40	4
findbugs	annotation	0	0	0	0	0	0
jetbrains	annotation	2,310	138	1,596	98	115	20
android	annotation	12,003	5,704	7,013	3,414	910	464
androidx	annotation	139,933	20,593	86,212	13,811	599	401
kotlin contracts	others	-	1	-	1	-	1

own a large percentage of the overall use in each category. Additionally, it's 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.

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

Lastly, Table 4.6 presents the frequency of each contract type. Once again, we have distinct results for Java and Kotlin. In Java, we found 63.73% of the classified instances in the last versions to be pre-conditions, 23.19% are post-conditions, and only 13.08% are class invariants. These results align with other empirical studies on contracts (Chalin, 2006; Schiller et al., 2014; Dietrich et al., 2017) that show a clear bias towards pre-conditions. However, Kotlin's results are very

Table 4.4: Gini coefficient by category.

Category	Java	Kotlin
assertion	0.70	0.71
API	0.80	0.37
annotation	0.88	0.76
CRE	0.77	0.67
others	-	1.00

Table 4.5: Top five applications using contracts (second versions only) by category.

Category	Applications
assertion	K1rakishou-Kuroba-Experimental (378), a-pavlov-jed2k (314), abhijitvalluri-fitnotifications (143), thundernest-k-9 (114), mozilla-mobile-firefox-android-klar (95)
CRE	redfish64-TinyTravelTracker (1,036), nikita36078-J2ME-Loader (690), abhijitvalluri-fitnotifications (561), lz233-unvcode-android (561), cmeng-git-atalk-android (447)
API	wbaumann-SmartReceiptsLibrary (534), alexcustos-linkasanote (318), BrandroidTools-OpenExplorer (69), oshepherd-Impeller (33), MovingBlocks-DestinationSol (30), inputmice-lttrs-android (24)
annotation	MuntashirAkon-AppManager (5,957), Forkgram-TelegramAndroid (5,552), Telegram-FOSS-Team-Telegram-FOSS (5,549), MarcusWolschon-osmeditor4android (4,393), NekoX-Dev-NekoX (4,032)
other	zhanghai-MaterialFiles (1)

different from this expected preference hierarchy. From the classified instances of the applications' last version, we found 38.82% to be post-conditions, 31.73% class invariants, and 29.44% pre-conditions. This leads to the conclusion that Kotlin developers tend to favor post-conditions over any other type, while pre-conditions come at the last position. We also highlight that according to the classification described in subsection 3.3.2, in our study, only annotations may be classified as post-conditions or class-invariants. This means that in Kotlin, there is a higher number of annotations associated with the method's return values and class properties than with the method's parameters. To further confirm this statement, we retrieved the top 100 applications with higher usage per type, which is represented in Figure 4.2 with the outliers hidden. Once again, through both plots, we see the different preference hierarchies between Java and Kotlin.

Finding 4: Java and Kotlin practitioners display different tendencies when it comes to the contract type. In Java, there is a clear bias towards pre-conditions, while in Kotlin, post-conditions are the most frequent type.

Although we can not provide a reason for this finding with certainty, we argue that the difference in practitioners' bias for each type reported in Table 4.6 could stem from different behavior patterns which are demonstrated in Table 4.7 and Table 4.8. These tables list the ten most occurring constructs for each type in the last versions of Java and Kotlin applications. To create these tables, we followed the classification described in subsection 3.3.2; hence, for example, although there are 2,217 instances of *JavaAssert* in Java, these were not included in the list since the analysis tool doesn't classify asserts by type.

Table 4.6: Number of contracts found in the dataset by type.

	contracts (all ver.)		contracts (2nd ver.)		applications	
Type	Java	Kotlin	Java	Kotlin	Java	Kotlin
pre-condition	120,671	9,203	72,160	5,744	989	349
post-condition	41,764	11,490	26,253	7,575	829	435
invariants	23,836	9,122	14,811	6,190	643	348
not classified	3,584	3,893	2,267	2,394	279	202

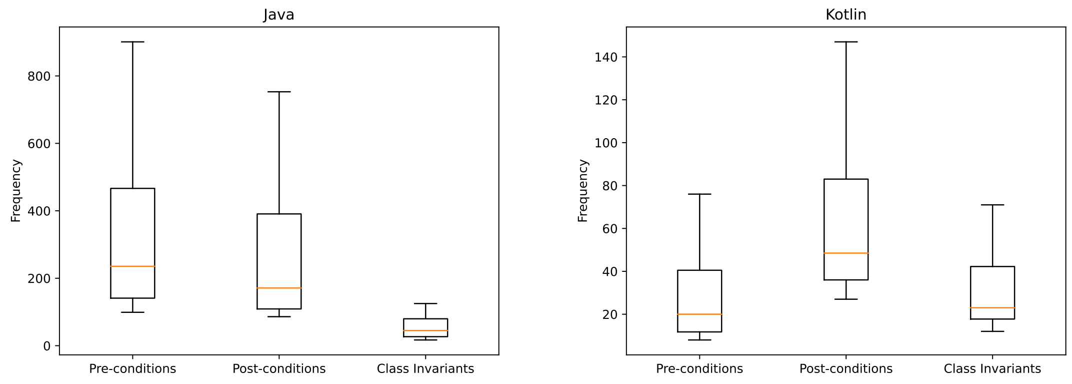


Figure 4.2: Comparison of the distribution of the identified contract types top 100 applications with higher usage per type for Java and Kotlin.

Table 4.7: The top 10 most frequent constructs per type in the last versions of Java applications.

Pre-conditions	Post-conditions
AndroidXNonNull (36,031)	AndroidXNonNull (10,359)
AndroidXNullable (14,983)	AndroidXNullable (5,954)
CREIllegalArgumentException (7,663)	AndroidSuppressLint (3,125)
CREIllegalStateException (3,232)	AndroidTargetApi (1,243)
CRENullPointerException (2,230)	AndroidXRequiresApi (732)
GuavaPreconditionNotNull (1,021)	AndroidXWorkerThread (551)
JSR305NonNull (860)	AndroidXKeep (398)
AndroidXStringRes (660)	AndroidXCallSuper (380)
CREIndexOutOfBoundsException (656)	AndroidXUiThread (326)
JetBrainsNotNull (612)	JSR305NonNull(322)

By comparing the two tables, we draw distinct behavior patterns between the two languages. In the Kotlin constructs reported by Table 4.8, none of the top ten most popular constructs relates to null-checking. But, in Java's instances reported in Table 4.7, 82.03% of pre-conditions and 71.12% of post-conditions are associated with null-checking. In this number, we are not considering potential *CREIllegalArgumentException* and *CREIllegalStateException* that could be associated with null-checking since this would require analyzing the condition present at the *if-statement*. This confirms a lack of expressiveness in the contracts specified by Java practitioners, with most being associated with null-checking, which aligns with previous studies (Schiller et al., 2014; Estler et al., 2014).

This contrast in null-checking contracts between Java and Kotlin is easily explained by the languages' different takes on nullability. In Kotlin, contrary to Java, regular types are non-nullable by default; therefore, in most cases, practitioners don't 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.

Finding 5: In Java applications' last versions, at least 77.72% of pre-conditions, 65.63% of post-conditions, and 61.24% of class invariants are related to null-checking. In the case of Kotlin, we found only about 3.12% of pre-conditions, 6.00% of post-conditions, and 0.57% of class invariants to be performing null-checking.

We have also looked into whether there was any relation between the number of contracts in the last version and any of the GitHub metrics from Figure 4.1. However, no meaningful correlation was found.

Table 4.8: The top 10 most frequent constructs per type in the last versions of Kotlin applications.

Pre-conditions	Post-conditions
AndroidXStringRes (1,142)	AndroidSuppressLint (2,289)
CREIllegalStateException (772)	AndroidXVisibleForTesting (1,663)
CREIllegalArgumentException (748)	AndroidXRequiresApi (720)
AndroidXColorInt (523)	AndroidXWorkerThread (638)
AndroidXDrawableRes (425)	AndroidXMainThread (441)
AndroidXAttrRes (255)	AndroidXCallSuper (319)
AndroidXColorRes (195)	AndroidXColorInt (237)
AndroidXIdRes (184)	AndroidTargetApi (205)
UCREUnsupportedOperationException (116)	AndroidXUiThread (195)
AndroidXFloatRange (80)	AndroidXAnyThread(184)

4.3 RQ2: How does contract usage evolve in an application?

In the second research question, we proposed to provide insights into how contract usage evolves in an application, namely on whether the numbers of contracts increase from the application's first to the last version.

Table 4.9 presents the number of contracts in the first and second versions by category. In general, we conclude that for most cases, the number of contracts in each category increased from the first to the last version. The only categories where the number decreased were the *Apache's Commons Validate* and in *JSR305 annotations package* for Java. The decrease in JSR305 usage could be explained by it currently being in a dormant status, or in other words, with no activity since 2017.

Additionally, to provide a clear view into this question, we computed some metrics to understand how the increase in the program's size relates to the number of contracts. Those metrics are listed in Table 4.10, including the average and median values for the number of methods, the number of contracts, and the ratio between both (contracts per methods number) for the first and second versions.

First, the table shows there was an average increase of about 109.936 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 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 -0.0057, and the median is -0.0012. Although the values are very small, we conclude that the number of methods

Table 4.9: Contract elements by type in the first and last version.

		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	1,133	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	53,721	6,782	86,212	13,811
kotlin contracts	other	-	0	-	1

increases significantly more than the number of contracts.

Finding 6: Although the total and average numbers of contracts increase while its median decreases by a small factor, we conclude that applications that use contracts continue to use them. Still, the number of methods increases by a greater scale than the contracts number.

Similarly to our study, Dietrich et al. (2017) also found that the median value of the ratio does not change much. Still, while we reported a decline between the first and last versions (0.029 to 0.022), they found a rise (0.021 to 0.023). This means that although both studies show general stability related to contracts usage, contrary to Dietrich et al. (2017) we were not able to find a positive correlation between the increase in the number of methods and the increase in the number of contracts.

Table 4.10: 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	310	972,037	356	1,081,973
contracts count	7	64,938	6	79,658
contract-to-method ratio	0.029	0.061	0.022	0.055

4.4 RQ3: Are contracts used safely in the context of program evolution and inheritance?

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 breaks, namely when *preconditions are strengthened* or *postconditions are weakened*. This process was described in more detail in subsection 3.3.3 and subsection 3.3.4.

It's 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. Nevertheless, Table 4.11 and Table 4.12 still provide valuable insights into the safety of contract usage and evolution.

First, let's consider Table 4.11, which displays the frequency of each *evolution pattern* in the context of *program evolution*. At first glance, we may be rushed to conclude that specifications are generally stable since the most frequent pattern is when a contract remains *unchanged* from the first to the second version. Unfortunately, this is not true since the occurrences of contract changes make up more than 50% of the patterns found. Still, overall, most of the changes are non-critical ones — including *minor changes*, *pre-conditions weakening*, and *post-conditions strengthening* — which is a positive finding. Less optimistic is that the second most common pattern is the case of *pre-conditions strengthening*, one of the two cases that potentially offers risk. In summary, although many contracts remain unchanged and most changes are not critical, we still found many occurrences that can lead to potential breaks.

Finding 7: There are instances of unsafe contract changes while the program evolves, particularly cases of pre-conditions strengthening.

Last, we look at Table 4.12, which presents the results found for *evolution patterns* in the context of *inheritance*. We observe that the *pre-conditions 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 low occurrences of *post-conditions weakening* when compared to the other classifications. Also, compared to the reports from Dietrich et al. (2017), our results indicate a greater ratio of *pre-conditions strengthening* per pre-conditions found.

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

Table 4.11: Contract evolution in the context of program evolution.

Evolution	Critical	Count
unchanged	no	32,070
minor change	no	199
pre-conditions weakened	no	13,870
post-conditions strengthened	no	9,906
pre-conditions strengthened	yes	20,870
post-conditions weakened	yes	5,461
unclassified	?	8,307

Table 4.12: Contract evolution in the context of inheritance.

Evolution	Critical	Count
unchanged	no	158
minor change	no	1
pre-conditions weakened	no	3
post-conditions strengthened	no	71
pre-conditions strengthened	yes	232
post-conditions weakened	yes	0
unclassified	?	145

Chapter 5

Conclusion

5.1 Summary

Object-oriented programming is an industry-standard that supports most software today. One of its core principles is the division of work between different reusable components where clients use services provided by suppliers. Still, this focus on creating modular components that interact with each other brings challenges when designing reliable software due to potential misunderstandings between clients and suppliers and the propagation of errors through the interaction. Over the years, many authors have proposed a systematic application of Design-by-Contract, where interaction between modules is defined through formal specifications to reduce errors.

This work presents a large-scale empirical study of the *usage* and *evolution* of contracts in Android applications. We evaluated 2,390 open-source Java and Kotlin applications. Since we analyzed two different versions for most applications, our dataset comprised 4.192 application-version pair which translated into 327,690 compilation units. Based on source code static analysis, we extracted and classified contract occurrences from that dataset. We also identified and classified contract changes according to evolution patterns in the context of *program evolution* and *inheritance*. The proposed dataset creation pipeline and the analysis tool enable an easily replicated experiment.

Overall, many of this work's findings confirmed already-known practitioners' practices and preferences presented in previous empirical studies. Nevertheless, the focus on Android programs made it possible to acquire particular insights into this type of application and distinct usage patterns between the Java and Kotlin languages. The section 5.2 summarizes the answers to the research questions according to the findings. Additionally, from these results, we can extract implications in the form of recommendations for practitioners, tool builders, and researchers, presented in section 5.3.

5.2 Answers to Research Questions

The answers to the research questions proposed in subsection 1.3.1 are summarized in the following way:

RQ1. How and to what extent are contracts used in Android applications?

The results show that the consistent and meaningful use of contracts is concentrated in a small number of applications. Still, when applications use contracts, annotation-based approaches are the most frequent in Java and Kotlin, where the *androidx.annotation* package is the most popular. This study found different types and preferences between the two languages. While in Java, 63.73% of the classified instances are pre-conditions, Kotlin programs display a more equally distributed selection with 38.82% post-conditions at the top. We also found that more than 50% of the classified contracts in Java are related to null-checking, extremely high compared to Kotlin's numbers of less than 10%.

RQ2. How does contract usage evolve in an application?

After comparing the number of contracts in both versions of the applications, we found that, on average, the number of contracts increased. Still, this was caused by some outliers that increased its usage intensively, driving up the average. In fact, the median value decreased. Furthermore, after computing the contract-to-method ratio, we found that this ratio decreased between versions — a median decrease of -0.0057 and an average decrease of -0.0012. Although by a residual factor, we observed that the number of contracts declined as the program grew.

RQ3. Are contracts used safely in the context of program evolution and inheritance?

The analysis tool builds different records of contracts appearing in both versions of the applications and contracts present in inheritance relations. After categorizing each diff record, we found that contract changes are widespread, meaning a lack of specifications' stability. From those changes, pre-conditions strengthening is the most classified pattern. These results clearly show a potentially unsafe use of contracts that may lead to client breaks.

5.3 Recommendations

From the findings presented in this work, we were able to derive implications for practitioners, researchers, tool builders, and academia in general. These implications can be translated into recommendations to increase and improve Design-by-Contract usage.

Recommendations for practitioners. Although the academic literature recommends adopting contracts to build more reliable software, this study shows that most practitioners do not use contracts. Additionally, Java practitioners display an obvious bias for pre-conditions over any other type of specifications (RQ1), which should be tackled by promoting education on DbC principles and by new tools. Furthermore, practitioners, particularly for Java, should strive to write more expressive contracts than the commonly found null-checking ones. As described in section 2.1, this is extremely important because a program may be correct against a weak contract but fail when contracts are more expressive. Last, practitioners should be made aware of the potential risks of some contract changes in the context of program evolution and inheritance (RQ3).

Recommendations for researchers and tool builders. Due to the visible fragmentation of technologies and approaches to specifying contracts (RQ1), ideally, both Java and Kotlin standard libraries should be equipped with specialized constructs to specify contracts and with proper official documentation. Alternatively, researchers and tool builders must create libraries to standardize contract specifications in the Java and Kotlin languages. The presented results suggest that this library be built around annotation-based contracts, given its popularity among practitioners (RQ1). An annotation-based approach, where specifications are added to the program as metadata, is similar to Eiffel's approach, where the assertions don't obfuscate the method's implementation. Creating patterns to guide contract specifications could also raise DbC's safe and efficient usage (RQ3). Additionally, due to the lack of expressiveness in contracts (RQ1), tools to aid practitioners in deriving new contracts could offer a valuable contribution to creating stronger specifications. Another contribution could be a continuous integration plugin to detect contract violations in the context of program evolution and inheritance (RQ3).

Recommendations for academia. The presented findings prove that there is still a broad gap between the academic literature and the industry's actual use of contracts; where although literature proposes many advantages related to contract usage, in practice, only some applications use them significantly. More evidence and proof should be studied and presented to the industry to foster practitioners' adoption and to close this gap. Potential new research directions are given in section 5.5.

5.4 Limitations and Threats to Validity

In this section, we discuss limitations and potential threats to this study's validity and its results. Our study suffers from the same limitations presented by Dietrich et al. (2017) due to the similarity of the methodologies used.

Firstly, the evaluation only evaluated open-source projects since the proposed methodology is based on source code analysis, and it would have been challenging to have access in enough quantity to closed-source commercial code. Nevertheless, having commercial apps in our dataset would contribute to the richness of this study. Also, in the context of the dataset build process, the fact that some program's versions had to be resolved manually to determine the app's first and later versions could lead to wrongly ordered versions affecting the evolution results.

The *contracts extraction process* is complex due to the various approaches and constructs to express contracts in Java and Kotlin. This is relatively more challenging than if the evaluation studied Eiffel applications with its standard way of defining contracts that could be easily identified by the analysis tool. Therefore, our constructs list may be incomplete, and we may be missing some patterns. Additionally, a limitation of our tool is that it doesn't recognize custom APIs or custom annotations created by practitioners. Lastly, our tool follows a naïve approach to associate each construct to a type (pre-conditions, post-conditions, and invariants). We perform that association according to the libraries' documentation, which defines the construct's purpose. We are not considering that the practitioner may use the constructs outside the intended purpose. For example, while Google's *Guava* library clearly states that the *Preconditions* class should be used to check pre-conditions, a practitioner may be using it to check post-conditions.

In the *evolution* and *inheritance* studies, the analysis tool tries to classify contract changes according to some evolution patterns. Since this classification is mechanical and, again, due to the variety of constructs, we may not be able to classify all instances. This limitation is seen when, for example, a method in the first version has an annotation constraint which is later changed to an assert in its second version. The analysis tool cannot understand whether the contract was strengthened or weakened. Also, in the *inheritance* study, the fact that we don't consider the libraries used by each application (due to size and time constraints) can result in an under-reporting. For example, the application may be overriding methods provided by one of its imported libraries which would not be considered by the analysis tool.

5.5 Future Work

Despite this work fulfilling its stated goals, the proposed solution presents some limitations described in section 5.4, which can be improved. Additionally, further research can be conducted to achieve more insights into how the industry uses contracts.

Firstly, we identify potential improvements to the analysis tool's capabilities:

- *Support for other languages.* This work added support for Kotlin source code to a pre-existing tool that could only investigate contracts in Java programs. Still, further work can be done to increase this tool's capabilities by adding new languages.

- *Improve contract classification by type.* In this work, CREs and API contracts were classified as pre-conditions according to the respective library's documentation or grey literature that states their intended purposes to validate arguments. Still, developers could use those constructs outside their intended purpose, resulting in a wrong classification. Additionally, the proposed tool does not classify asserts as pre-conditions or post-conditions, which may be improved.

Now, we list potential research directions that could provide new information about contracts and their use:

- *Understand what makes a program use contracts significantly.* In this study, we understood that some programs use contracts intensively, and others do not use them meaningfully. Additionally, some programs increase their usage from the first to the last version while others decrease it. It would be important to find whether common characteristics or patterns are shared between programs that use many contracts. Some aspects to consider could be the team size, seniority of team members, business domain, code metrics, and others...

- *Understand whether a correlation exists between projects that use contracts and those with higher-quality code.* An crucial insight would be to determine whether practitioners that use contracts intensively are also the ones who write higher-quality code. These metrics may include Cyclomatic Complexity (MVG), coupling between object classes, and cohesion in methods, among others...

- *Understand the challenges and obstacles to specifying contracts.* Since this study found that only some applications use contracts intensively, it would be valuable for this domain to discover the challenges and obstacles that developers face that hamper DbC adoption. This study should be conducted with practitioners to understand the causes and draw actions to solve those challenges.

Bibliography

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.

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.

M. Aniche. *Effective Software Testing. A Developer's Guide.* Manning, Shelter Islands, 2022. ISBN 9781633439931.

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. ISBN 9781450341394.

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, 2002a.

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, 2002b.

D. Carvalho, R. Hussain, A. Khan, M. Khazeev, J. Lee, S. Masiagin, M. Mazzara, R. Mustafin, A. Naumchev, and V. Rivera. Teaching programming and design-by-contract. In *The Challenges of the Digital Transformation in Education. Proceedings of the 21st International Conference on Interactive Collaborative Learning (ICL2018). Advances in Intelligent Systems and Computing (AISC 916)*, volume 1, pages 68 – 76, Cham, Switzerland, 2020.

M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. Heartbleed 101. *IEEE Security & Privacy*, 12(4):63–67, 2014.

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.

P. Chalin. *Are practitioners writing contracts?*, pages 100 – 113. Springer, Berlin, Germany, 2006. ISBN 978-3-540-48265-9.

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.

M. Danisovszky, T. Nagy, K. R´ep´as, and G. Kusper. Western canon of software engineering: The abstract principles. In *2019 10th IEEE International Conference on Cognitive Infocommunications*, pages 153–156, 2019.

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. ISBN 978-3-95977-035-4.

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.

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

R. W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993. ISBN 978-94-011-1793-7.

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. ISBN 9781450394130.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

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.

M. Huisman and R. E. Monti. Teaching design by contract using snap! In *2021 Third International Workshop on Software Engineering Education for the Next Generation (SEENG)*, pages 1 – 5, Piscataway, NJ, USA, 2021.

J.-M. Jazequel and B. Meyer. Design by contract: the lessons of ariane. *Computer*, 30(1):129–130, 1997.

A. K. Jha and S. Nadi. Annotation practices in android apps. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 132 – 42, Los Alamitos, CA, USA, 2020.

John Knight. Safety critical systems: Challenges and directions. In *Proceedings - International Conference on Software Engineering*, pages 547 – 550, 2002. ISBN 1-58113-472-X.

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.

G. Kudrjavets, N. Nagappan, and T. 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.

G. Le Lann. An analysis of the ariane 5 flight 501 failure-a system engineering perspective. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, pages 339–346, 1997.

R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003. ISBN 0135974445.

S. McConnell. *Code Complete*. Microsoft Press, 2 edition, 2004. ISBN 0735619670.

B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40 – 51, 1992. ISSN 0018-9162.

B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997. ISBN 978-0-13-629155-8.

Microsoft Learn contributors. Code contracts (.net framework), 2021. URL https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts. [Online; accessed 3-February-2023].

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.

G. J. Myers. *Software Reliability: Principles and Practices*. Willey, 1 edition, 1976. ISBN 0-471-62765-8.

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

P. Naur. Proof of algorithms by general snapshots. *BIT Computer Science and Numerical Mathematics*, 6(4):310–316, 1966.

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. ISBN 9781450327565.

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.

StatCounter Global Stats. Operating system market share worldwide, 2023. URL `https://gs.statcounter.com/os-market-share#monthly-202208-202209-bar`. [Online; accessed 3-February-2023].

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.

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

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.

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.

P. Xu and S. Xu. A reliability model for object-oriented software. In *2010 19th IEEE Asian Test Symposium*, December 2010.

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.

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.

Appendix A

List of Conditional Runtime Exceptions analyzed

Table A.1: List of exceptions analyzed in the CRE category.

CREs Constructs	
AndroidRuntimeException	MissingResourceException
ArithmeticException	NegativeArraySizeException
ArrayStoreException	NoSuchElementException
ArrayStoreException	NullPointerException
BufferOverflowException	ParcelFormatException
BufferUnderflowException	ParseException
ClassCastException	ProviderException
CompletionException	ProviderNotFoundException
ConcurrentModificationException	RejectedExecutionException
DOMException	SQLException
DateTimeException	SecurityException
EmptyStackException	TypeNotPresentException
EnumConstantNotPresentException	UncheckedIOException
FileSystemAlreadyExistsException	UndeclaredThrowableException
FileSystemNotFoundException	UnsupportedOperationException
IllegalArgumentException	WrongMethodTypeException
IllegalMonitorStateException	AcceptPendingException
IllegalStateException	AccessControllException
IncompleteAnnotationException	AlreadyBoundException
IndexOutOfBondException	AlreadyConnectedException
LSException	ArrayIndexOutOfBondsException
MalformedParameterizedTypeException	BadParceableException

MalformedParametersException	CancellationException
UnsupportedAddressTypeException	UnsupportedCharsetException
WritePendingException	ZoneRulesException
CancelledKeyException	PatternSyntaxException
ClosedDirectoryStreamException	StringIndexOutOfBoundsException
ClosedFileSystemException	ReadOnlyBufferException
ClosedFileSystemException	ReadOnlyFileSystemException
ClosedSelectorException	ReadPendingException
ClosedWatchServiceException	ShutdownChannelGroupException
ConnectionPendingException	StringIndexOutOfBoundsException
NonReadableChannelException	UnknownFormatConversionException
NonWritableChannelException	UnknownFormatFlagsException
NotYetBoundException	UnresolvedAddressException
NotYetConnectedException	UnsupportedTemporalTypeException
NumberFormatException	OverlappingFileLockException

Appendix B

List of API's methods analyzed

Table B.1: List of the methods analyzed from each API.

Annotations analyzed	

Apacha lang2 Validate	allElementsOfType()
	isTrue()
	noNullElements()
	notEmpty()
	notNull()

Apacha lang3 Validate	allElementsOfType()
	exclusiveBetween()
	inclusiveBetween()
	assignableFrom()
	isInstanceOf()
	matchesPattern()
	notBlank()
	validIndex()
	validState()

Guava Preconditions	checkArgument()
	checkState()
	checkElementIndex()
	checkPositionIndex()
	checkNotNull()

	checkPositionIndexes()

Spring Assert	doesNotContain()
	hasLength()
	hasText()
	notEmpty()
	noNullElements()
	isInstanceOf()
	isAssignable()
	state()
	isNull()
	isTrue()
	notNull()

Appendix C

List of Annotations analyzed

Table C.1: List of annotations analyzed per package.

	Annotations analyzed	
JSR305	@CheckForNull	@CheckForSigned
	@MatchesPattern	@Nonnegative
	@Nonnul	@Nullable
	@OverridingMethodsMustInvokeSupper	@ParametersAreNonnullByDefault
	@RegEx	@Signed
	@Syntax	@Syntax
	@Tainted	@Untainted
	@WillClose	@WillCloseWhenClosed
	@WillNotClose	@Guardedby
	@Immutable	@NotThreadSafe
	@ThreadSafe	
JSR303, JSR349	@Null	@DecimalMin
	@NotNull	@Size
	@AssertTrue	@Digits
	@AssertFalse	@Past
	@Min	@Future
	@Max	@Pattern
	@DecimalMax	
JetBrain	@Contract	@NotNull

	@Nullable	@PropertyKey
	@TestOnly	
IntelliJ	@BoxLayoutAxis	@CalendarMonth
	@CursorType	@FlowLayoutAlignment
	@FontStyle	@HorizontalAlignment
	@InputEventMask	@ListSelectionMode
	@PatternFlags	@TabLayoutPolicy
	@AdjustableOrientation	@Flow
	@Identifier	@TabPlacement
	@TitledBorderJustification	@TitledBorderTitlePosition
	@Language	@MagicConstant
	@Pattern	PrintFormat
	@PrintFormat	@RexExp
	@Subst	
FindBugs	@CheckForNull	@NonNull
	@Nullable	@PossiblyNull
	@FontStyle	@HorizontalAlignment
	@UnkownNullness	@CreateObligation
	@DischargesObligation	@CleanupObligation
Android	@AndroidSupressLint	@AndroidTargetApi
Androidx		
	@AnimatorRes	@AnimRes
	@AnyRes	@AnyThread
	@AnyThread	@ArrayRes
	@AttrRes	@BinderThread
	@BinderThread	@BoolRes
	@CallSuper	@CheckResult
	@ChecksSdkIntAtLeast	@ColorInt
	@ColorLong	@ColorRes
	@ContentView	@DimenRes
	@Dimension	@NotInline
	@DrawableRes	@FloatRange

@FloatRange	@FontRes
@FontRes	@FractionRes
@FractionRes	@GuardedBy
@GuardedBy	@HalfFloat
@IdRes	@InspectableProperty
@IntDef	@IntegerRes
@InterpolatorRes	@IntRange
@Keep	@LayoutRes
@LongDef	@MainThread
@MainThread	@MenuRes
@NavigationRes	@NonNull
@Nullable	@PluralsRec
@Px	@RawRes
@RequiresApi	@RequiresFeature
@RequiresPermission	@RestrictTo
@Size	@StringDef
@StringRes	@StyleableRes
@StyleRes	@TransitionRes
@UiThread	@VisibleForTesting
@WorkerThread	@XmlRes