How have iOS Development Technologies Changed over Time? A Study in Open-Source

Luciano Baresi, Giovanni Quattrocchi, Damian Andrew Tamburri
Politecnico di Milano
Milan, Italy
{name.surname}@polimi.it

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

ABSTRACT

iOS development has undergone a significant shift related to the introduction of the Swift programming language, conceived to replace or complement Objective-C. This may have impacted how iOS apps are developed, e.g., concerning User Interfaces (UI). This study investigates current trends in iOS app development, focusing on programming language preferences and UI framework adoption, by examining 140 open-source native iOS applications. The results indicate that 94% of the apps now integrate Swift code, showcasing its widespread acceptance since its 2014 inception, and also show different migration patterns. The paper also compares the established UIKit, a popular framework for building iOS user interfaces, with the recently introduced SwiftUI, which was introduced in 2019. Our analysis shows a strong uptake of SwiftUI, used in 64 of the apps in our dataset. Notably, for apps initiated after the introduction of SwiftUI, almost half were developed with SwiftUI from the start.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; **Programming languages**; **Mobile application development**; **Programming languages**.

KEYWORDS

iOS, Objective-C, Swift, Mobile Applications

ACM Reference Format:
Luciano Baresi, Giovanni Quattrocchi,, Damian Andrew Tamburri and Massimiliano Di Penta. 2024. How have iOS Development Technologies Changed over Time? A Study in Open-Source. In *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MOBILESoft '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3647632.3647988

1 INTRODUCTION

When we think of developing a mobile app, one of the first decisions to take is the identification of the operating systems of interest [19]. The number of choices is limited: one can develop *native* applications for either Android or iOS devices, or also adopt *cross-platform* frameworks to develop for both operating systems—and others—at the same time. The number of sold devices[1] would suggest to only focus on Android. However, the importance of iOS applications is witnessed by the fact that iOS was responsible for 67% of total app consumer spending in 2022[2].

These numbers motivate a detailed study of how the development of iOS applications has evolved over the years. Mobile operating systems are very prescriptive in terms of programming languages and frameworks developers can use, and iOS is no exception [3]. Originally, iOS imposed the use of Objective-C as a programming language and UIKit (Storyboard) as a framework for creating user interfaces. Since 2014, Apple suggested the use of Swift instead of Objective-C, and, since 2019, the use of SwiftUI instead of UIKit. After the introduction of the new languages and frameworks, all iOS versions have remained backward-compatible, and at the time of writing (iOS 17) one can still base the development of a new application, or the maintenance of an existing one, on the pair Objective-C/UIKit.

This coexistence leads to an emerging need for a deeper understanding of how the community has been adopting these new options, and how existing applications have evolved—or are evolving—to exploit them. Even if many popular iOS apps are not open-source, we conducted our analysis by considering the vast corpus of open-source iOS applications available on GitHub as the only source available. Our initial search revealed 1309 potential projects. After conducting a manual analysis, and filtering out projects not related to apps, abandonware, and other false positives, we identified a dataset of 140 significant open-source iOS apps to study the evolution of adopted technologies featuring both well-known apps (e.g., Firefox[3] and ProtonVPN[4]), as well as some emerging ones of the two million apps available on the App Store.

Our study focuses on three main aspects:

- The choice of the programming language, and in particular the transition from Objective-C to Swift.
- The patterns that governed Swift adoption, that is, the versions of the language that receive higher attention.
- The use of UIKit vs. SwiftUI, and the related implications.

This study provides insights into the evolving landscape of iOS app development, highlighting the adoption and impact of the Swift programming language and the integration of newer UI frameworks like SwiftUI. Our analysis reveals that 94% of the apps now integrate

MOBILESoft '24, April 14–15, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0594-6/24/04.
https://doi.org/10.1145/3647632.3647988

[1] https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/
[2] https://www.businessofapps.com/data/app-revenues/
[3] https://www.mozilla.org/en-US/firefox/browsers/mobile/ios
[4] https://protonvpn.com/download-ios

Swift code, and 64 apps—out of 140—use SwiftUI, with almost half of the apps created after its introduction adopting such UI technology.

The rest of the paper is organized as follows. Section 2 summarizes the key technologies for developing iOS applications. Section 3 define the study research question and analysis methodology. Section 4 reports and discusses the obtained results, while its threats are discussed in Section 5. Section 6 surveys the related work, and Section 7 concludes the paper.

2 BACKGROUND

This section provides background about pieces of technology—programming languages and user interface (UI) frameworks in particular—used to develop iOS apps.

2.1 Programming languages

Objective-C extends the C language with object-oriented capabilities, message passing, and a rich and dynamic type system [13]. Listing 1 shows an example of Objective-C code. The language requires developers to define classes and methods across header (`.h`) and implementation (`.m`) files shown in lines 1-4 and lines 6-15 respectively. Objective-C requires an explicit memory management, and is characterized by a peculiar syntax for method invocations, which uses square brackets and named parameters (as shown in lines 14-15).

As shown in Listing 2, Swift supports concise syntax and object-oriented capabilities that can be complemented with functional programming. Each version of Swift has emphasized safety, performance, and ease of use, reflecting its steady evolution to meet modern development needs [1, 14, 15].

```
1  // Objective-C MyClass.h file
2  @interface MyClass : NSObject
3  - (void)doSomethingWithParameter:(NSString *)
       parameter;
4  @end
5
6  // Objective-C MyClass.m file
7  @implementation MyClass
8  - (void)doSomethingWithParameter:(NSString *)
       parameter {
9      NSLog(@"Parameter: %@", parameter);
10 }
11 @end
12
13 // Objective-C method invocation
14 MyClass *myInstance = [[MyClass alloc] init];
15 [myInstance doSomethingWithParameter:@"Example"];
```

Listing 1: Objective-C class and method invocation

2.2 UI Frameworks

Native iOS apps have been using the UIKit framework for their GUI since the first iOS Software Development Kit (SDK) was released in 2008. UIKit is built around the Model-View-Controller (MVC) design pattern. This pattern segregates the app into three interconnected components, which increases modularity and, in general, maintainability.

UIKit requires that developers use `UIViewControllers` to manage the content of views (GUIs), which is typically loaded from

```
1  // Swift MyClass.swift file
2  class MyClass {
3      func doSomething(withParameter parameter: String
       ) {
4          print("Parameter: \(parameter)")
5      }
6  }
7
8  // Swift method invocation
9  let myInstance = MyClass()
10 myInstance.doSomething(withParameter: "Example")
```

Listing 2: Swift class and method invocation

dedicated XIB files. These XML files are created through a dedicated graphical editor with a drag&drop user interface, called Interface Builder. A `UIViewController` is responsible for materializing the GUI, populating it with data, interacting with the user, and modifying stored data if needed.

XIB files are XML Interface Builder documents used in the development of applications for Apple platforms. They serve as a blueprint for the user interface, allowing developers to design and layout views and view controllers in a graphical, WYSIWYG (What You See Is What You Get) environment. Each XIB file typically corresponds to a single screen's worth of content, and can define the properties and layout of various UI components.

Storyboard files—similar to XIB files—allow developers to define the entire flow of the screens of an application (known as scenes), and to visually define the transitions between them. In contrast, XIB files typically manage isolated UI components. Developers can lay out scenes (each associated with a `UIViewController`) and design transitions, known as *segues*, between these scenes to define the navigation flow of the app. The Storyboard also allows one to link UI elements to related code to enable programmatic control of these UI components. However, some developers prefer to define their GUIs and ViewControllers programmatically. This approach provides more control at the cost of a more complex development. UIKit, originally designed for Objective-C, can also be seamlessly utilized with Swift.

SwiftUI represents a paradigm shift in how developers create user interfaces. Unlike UIKit, which is imperative and requires developers to specify how GUI elements are to be displayed and managed, SwiftUI is declarative and focuses on what the GUI should contain [4]. Also, SwiftUI enables developers to define GUI elements and their behaviors entirely in (Swift) code. A live preview canvas provides immediate visual feedback. This integration removes the disconnection often experienced with UIKit, where the layout defined in a Storyboard might not perfectly match the final application. SwiftUI naturally supports the Model-View-ViewModel pattern [12]. In the MVC pattern used by UIKit, Controllers tend to become "massive" as they accumulate many responsibilities, such as updating the View with changes from the Model, handling user input, and navigating between screens. In contrast, the ViewModel in SwiftUI serves a similar but more focused role. It is responsible for transforming Model data into values that can be displayed on the View. Unlike Controllers, ViewModels do not deal with user input directly; instead, they provide bindable properties that SwiftUI's

Views can react to. This leads to a more modular and testable architecture, as ViewModels are typically simpler and more decoupled from the UI layer than Controllers [12].

While SwiftUI requires iOS 13 or newer, developers have the flexibility to integrate SwiftUI views into existing UIKit-based projects and vice versa, thanks to the interoperability between the two frameworks.

3 STUDY DESIGN

The *goal* of this study is to analyze native open-source iOS apps, to understand how they change the used technology during their lifetime. The study focuses on adopted programming language(s), and UI frameworks. The perspective is of *researchers* who want to understand how apps evolve when the underlying technology changes. The *context* consists of 140 open-source, native iOS apps. The study aims to address the following research questions:

RQ_1: *How do the apps' codebases change over time in terms of used programming languages?*
RQ_2: *How do the apps' UI implementation framework change over time?*

In the following, we describe the methodology we followed to address the research questions formulated above.

First, we selected the iOS apps to study. Then, after cloning their repositories, we analyzed their evolution history to identify the programming languages being used (i.e., Objective-C and/or Swift), and the adopted UI framework (UIKit vs. SwiftUI).

3.1 Selection of the Apps to be Studied

At first, we tried to search apps by querying GitHub and GitLab using the main programming language (Swift, Objective-C) or topic (e.g., *swift*, *ios-app*, or *ios*) as criteria. This approach was unsuccessful as it led to misleading results: it returned several irrelevant repositories, while failing to retrieve many relevant apps.

Therefore, we decided to leverage an existing, collaborative, and curated list of open-source iOS apps available on GitHub[5]. The list contained, at the time of download, 1,309 different repository references (pointing to GitHub and other hosts). Using the search APIs of the respective hosting services, we obtained, for each repository, metadata such as the creation date and the number of stars.

We then started to filter the set of repositories using multiple criteria:

App: The repository must contain an actual application, and not libraries, extensions, SDKs, etc. This analysis has been manually conducted by looking at the README files. This activity filtered out 82 repositories and we retained 1227 apps.

Distributed in the Store: An app may be distributed, that is, it is made publicly available, through the only iOS store, the App Store. This limits the analysis to real-world apps that underwent the needed quality controls for publishing them in the store [5]. To apply this filter, we first searched the README files for references to the App Store (links, badge *Available on the App Store*), and then we searched the App Store for the apps. 956 apps, out of 1227, were not available on the App Store at the time of checking, leaving us with 271 apps.

[5]https://github.com/dkhamsing/open-source-ios-apps

Native: We excluded hybrid apps and only considered native apps developed using Objective-C and/or Swift. This is because we are interested in analyzing the evolution of the app concerning the usage of such languages and related UI frameworks. 233, out of 271 distributed apps, turned out to be native apps. To filter them, we checked the programming language metadata provided by GitHub.

Adequate history: To provide meaningful historical data, we only chose apps whose history spans over the time period in which the iOS technology—i.e., programming language and UI framework—changed;

Note that we did not rank (nor select) apps based on their number of stars, as this is a criterion more related to the app's popularity (and not necessarily suited for certain studies [2]) than to characteristics apps should have (as the ones mentioned above).

3.2 Selection of relevant commits

To analyze the evolution history of studied apps, we decided to identify a sample of commits over their evolution history. This has been done because the data extraction for each commit was relatively demanding, and it was therefore too expensive to analyze all commits.

At first, we thought about using release tags. Unfortunately, these were not consistently used across all the analyzed apps, and their usage over the years was very uneven. Therefore, we chose to select a sample of a maximum of 12 commits for each repository per year (equally spaced in terms of time) taken from the main branch of the app evolution history. For years with fewer than 12 commits, we retained all the available ones.

We stored all the names and references of the 140 apps in a MATLAB cell array: rows identify repositories and columns repository names, expressed as `owner_name/repository_name`, creation date, latest edit date, number of stars, repository URL and reference to the relative commit table. Commit tables store the data of filtered commits. Each app has its commit table, where the columns list commit hashes and the corresponding commit date.

In the following sections, when we say that an action was performed *for each commit of each app*, we mean that we iterated over the commit table rows of each app, checking out the corresponding repository at the commit, and storing derived data in appropriate columns of the table by using Algorithm 1.

Checking out a commit means reverting the repository state (i.e., its files) to how it was when the commit was made. This allowed us to analyze how the repository changed over time. Analysis software can then access the repository in any past state and return results about its features at that time.

Algorithm 1 Data gathering in all commits of each app

```
cd repository clones directory
for app in apps do
    cd app.name
    for commit in app.commits do
        git checkout -f commit.name
        obtain data about the repository
        store the obtained data as a new field of commit
    end for
end for
```

3.3 Programming Languages

To study used programming languages, we first checked out each sampled commit. Then, we computed, for each of these commits, the distribution of lines of code for each programming language. Note that, besides Objective-C and Swift, we also analyzed the amount of C and C++ code, because Objective-C extends from C, and allows for the inclusion of both C and C++ code.

To obtain data about programming language distribution in different commits of different apps, we computed the number of source code lines (LOC) in either Objective-C or Swift. To this aim, we leveraged the open-source command-line tool `cloc`[6], which, in turn, is based on `SLOCCount`[7]. Given a file, it identifies its underlying programming language(s): it does not only consider file extensions, but it also adopts different heuristics. The tool excludes commented lines from LOC counting.

To exclude files that are part of the repositories but not of the actual apps, we used `cloc` only on the directory that contains the main `.xcodeproj` package of the project. As we wanted to exclude potential files that belong to dependencies (i.e., that contain the so-called *vendored* code), and make the returned metrics as adherent as possible to the actual apps, we also added the argument `--exclude-dir=fastlane,Carthage,Vendor,vendor,-Pods,.build`, so to ignore the directories that often contain those files (i.e., directories such as `/Pods` and `/Carthage`).

3.4 UI Frameworks

As for the used UI frameworks, we found that the most effective way to study the evolution of their adoption is to count the number of UIKit and SwiftUI elements in each commit. While this accounts for elements from heterogeneous sources, it still gives an idea of the proportion of different types of UI elements in the app. To obtain the number of UIKit and SwiftUI elements, we adopted a different approach for each element.

To count the number of the different types of *UIKit Views and ViewControllers specified in Swift*, we used a customized version of the open-source tool *Sitrep*[8] on `.swift` files. Given an Xcode project, Sitrep returns a JSON structure that contains a high-level view of the Swift code in the project. Besides various code metrics, it also provides the number of UIViews, UIViewControllers, and SwiftUI view elements.

Since the original version of Sitrep counted as Views only the classes that inherit from class `UIView` and as ViewControllers those that inherit from class `UIViewController`, we modified it to match all classes that inherit from a class whose name starts with `UI` and ends with `View` being that all the UIView subclasses in UIKit follow this pattern (e.g., UIImageView). Similarly, to get the number of ViewControllers, we counted all classes that inherit from a class having a name that begins with `UI` and ends with `Controller`. The complete list of View and ViewController classes is available in the UIKit documentation [6].

Since, to the best of our knowledge, there is no tool equivalent to Sitrep for Objective-C, to count the number of the different types of *UIKit Views and ViewControllers specified in Objective-C*, we opted for using a series of heuristics to analyze the code. We counted as ViewControllers all classes that inherit from a class whose name begins with `UI` and ends with `Controller` and as Views all classes that inherit from a class whose name starts with `UI` and ends with `View`.

Since SwiftUI Views are structs that conform to protocol `View`, we counted *SwiftUI Views* by exploiting the feature available in Sitrep.

To conclude, we also considered *storyboards*, that is, files with extension `.storyboard`. They encode in XML the overall structure of an app, that is, its Views, ViewControllers, scenes, constraints, and links to the source code. This means that visiting the XML tree allows for counting the occurrences of the different element types, whose names match patterns related to UIKit and ViewController classes.

3.5 Dataset

All the aforementioned analysis tasks led to the dataset described in the following. The time span of considered repositories ranges between 0.25 and 14.35 years (4.74 years on average). Also, we have 131 repositories, out 140 (94%), with their last commit after Jan 1, 2020, while 100 apps (71%) —a subset of the former— have their last commit after Jan 1, 2022. The apps with longer time spans can offer data about the evolution of certain features. Instead, the apps with a shorter time span can show how the adoption of the different technologies has changed. Note that we also included apps whose last commit was published earlier because they are important to get meaningful data about the evolution of certain aspects.

Considered apps belong to diverse categories, ranging from (i) very popular open source apps such as Mozilla Firefox or VideoLAN VLC; (ii) security- and privacy-focused apps such as Signal, ProtonMail, and DuckDuckGo; (iii) country-specific apps, e.g., COVID-19 tracing apps; and (iv) educational apps, e.g., popular apps like Unwrap, or apps for learning Swift. The vast majority of the selected apps are distributed on the App Store for free, which is expected as they are open source.

4 STUDY RESULTS

This section reports the results of our study, answering the research questions formulated in Section 3.

4.1 RQ_1: Adoption of programming languages

To address RQ_1, we first analyzed the programming languages used to develop the 140 apps along their evolution history.

Figure 1 shows how the usage of the four languages evolved over time. We observe that C and C++ make up a relatively small fraction of the code base, largely attributed to legacy code that has persisted in the count. As expected, the most used languages are Objective-C and Swift. For Objective-C, we observed an increasing usage up to mid-2018. Since then, its expansion slowed down, with only a marginal increase in recent years. Swift entered the scene in late 2014 and only began to gain significant traction around 2017. Its adoption surged rapidly, catching up to Objective-C by early 2019 and currently being used threefold in comparison. This surge is due to the emergence of Swift-exclusive projects and a notable shift from Objective-C to Swift in many preexisting applications.

[6] https://github.com/AlDanial/cloc
[7] https://dwheeler.com/sloccount/
[8] https://github.com/twostraws/Sitrep

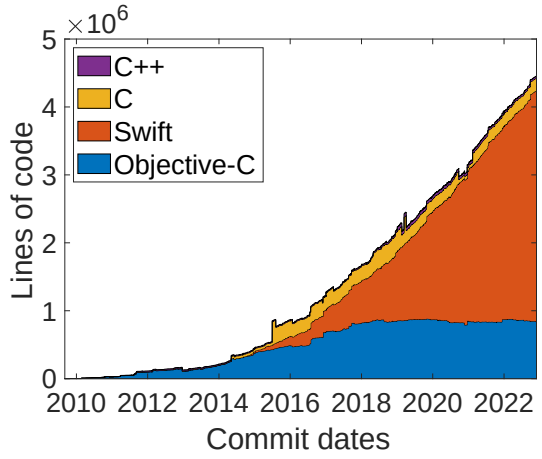


Figure 1: Cumulative adoption of languages.

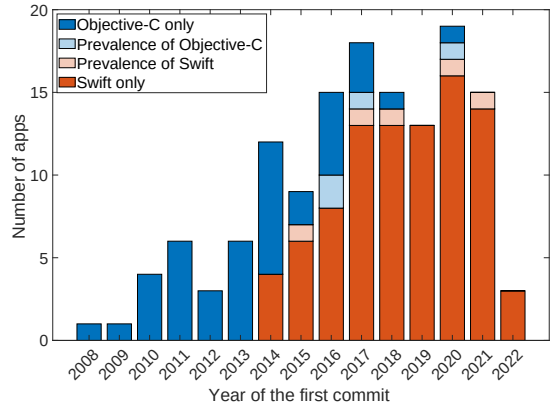


Figure 2: Language breakdown based on apps' first commit.

If we consider that Swift usually requires fewer lines of code than Objective-C to implement the same piece of functionality [10], the shift towards Swift in iOS app development is even more notable. Swift surpassed Objective-C in usage five years ago: its adoption by the developer community comes from its modern features. The use of C and C++ remains largely related to existing dependencies that are yet to be updated to the newer programming paradigms.

Finding #1: The evolution of language usage in iOS app development reflects a significant transition towards Swift. The data shows a steady decline of Objective-C since mid-2018, while Swift's adoption has surged, particularly since 2017. This trend denotes the developer community's preference for Swift's modern features and efficiency. Although Objective-C remains in use (largely for legacy code) the current trajectory suggests a continued and increasing dominance of Swift in the iOS development landscape.

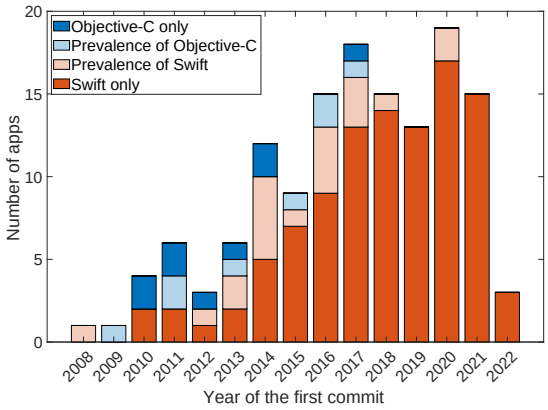


Figure 3: Objective-C, Swift breakdown for the first commit of each app, with data from the latest commit

4.2 RQ$_1$: Migrating from Objective-C toward Swift

To better understand the transition from Objective-C to Swift we analyzed the commits of each app. We divided the commits into four classes based on the language percentages of the code they contain:

- **Swift only**: commits with more than 98% of code written in Swift.
- **More Swift**: commits with between 50% and 98% of code written in Swift.
- **More Objective-C**: commits with between 50% and 98% of code written in Objective-C.
- **Objective-C only**: commits with more than 98% of code written in Objective-C.

Figure 2 shows, for each year, the type of the first commit for the different apps. Before 2014, as expected, all new apps were written using Objective-C since Swift was only introduced in 2014. During 2014-2017, Objective-C still had noticeable relevance in developers' choices. We noticed however that the number of Swift-only apps kept increasing. The reluctance to abandon Objective-C can be motivated by the fact that Swift was still too young to be adopted, and developers chose to use the "standard" programming language, which was better documented and supported at the time, instead of learning a new one. In its first years, Swift, and its related APIs, underwent many renaming and refactoring activities, and significant parts of the code became deprecated and required manual updates. Starting from 2018, most new apps have been built using Swift only.

To better analyze how the transition occurred, Figure 3 shows the type of the apps' last commit arranged according to year of their first commit (e.g., the third column reflects the types of last commits of apps created in 2010, specifically 2 *Swift Only* and 2 *Objective-C only*). This figure is especially useful to highlight the blends across patterns reported in our study. All the apps started after 2017 (not included) have their last commit of type either *More Swift* or *Swift*

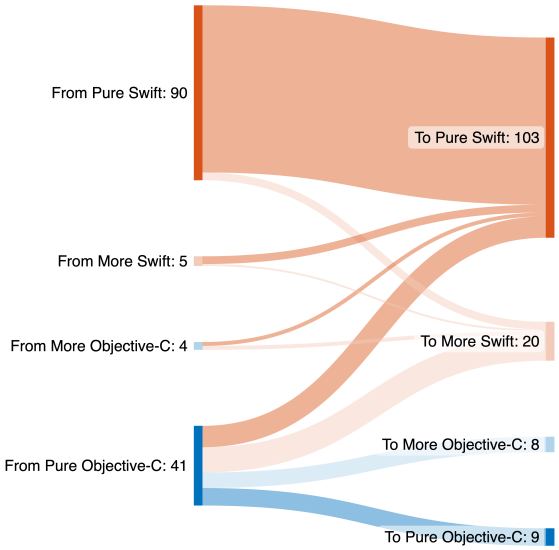


Figure 4: Flows between programming languages.

only. Notably, the eldest application considered, started in 2008, has a last commit of type *More Swift*.

Figure 4 details how each application evolved from the first to the last commit. Overall, out of 140 apps, 90 were initially entirely written in Swift, and 5 with a prevalence of Swift. Among the 45 applications that started with a prevalence of Objective-C, only 17 have the latest commit with a prevalence of Objective-C. Among the 41 applications that were initially written entirely in Objective-C, 32 were at least partially migrated to Swift.

> **Finding #2:** Our analysis indicates a Swift increasing adoption in iOS development. Before 2014, all new apps were Objective-C based, reflecting its established status. Between 2014 and 2017, developers cautiously transitioned to Swift, deterred initially by its novelty and evolving nature. After 2017, the trend shifted significantly towards Swift, with most new apps exclusively using it. This transition is further emphasized in older apps, where many have adapted to include Swift, indicating a broad acceptance and migration towards this (modern) language.

We then manually inspected each application to understand how the migration occurred over time. For example, *Wire*[9] (Figure 5a) started with a prevalence of Objective-C code and gradually replaced it almost completely with Swift code (at around commit 55). *PIA VPN*[10](Figure 5b) relied on a minor quantity of Objective-C code since the beginning. At around commit 20 it also removed that small portion.

We observed an opposite behavior in *susi.ai* [11](Figure 5c). The app was initially written only in Swift and introduced a small quantity of Objective-C code at commit 17, and increased it at commit 28. The Objective-C belongs to a chart drawing library written in Objective-C, whose code was included directly in the app source code.

WordPress[12](Figure 5d), with its first commit in July 2013, began adding Swift code in September 2014 (at around commit 80), just after its first release. Since then, Swift was the go-to for the new components but existing ones have been kept in Objective-C.

The trajectory of *Simplenote*[13], as shown in Figure 5e, starts in August 2016 with a significant amount of Objective-C code. The app saw a gradual decline in Objective-C usage as Swift code was introduced and eventually became the majority by October 2020. This evolution is indicative of a broader trend where Swift is becoming the favored language for iOS app development.

Lastly, *Blurry*[14], presented in Figure 5f, demonstrates a consistent approach to language use. Since its initial commit performed in June 2017, the application has kept a stable presence of Objective-C code alongside dominant Swift code, illustrating the app's commitment to Swift while leveraging the necessary aspects of Objective-C.

The shift towards Swift is a significant trend observed among the examined iOS applications. A majority of them (61%) were developed exclusively in Swift from their inception. Moreover, 78% of the applications that began with only Objective-C code eventually adopted Swift to some extent. Within this subset, 34.3% completely transitioned to Swift, while 65.7% now features a mix of both Objective-C and Swift. This shift commonly involves converting existing Objective-C code to Swift or incorporating new Swift code alongside the existing Objective-C base.

Through a manual analysis, conducted by three authors independently (and with classification made by discussing all disagreement cases), we identified six distinct patterns in the transition from Objective-C to Swift in iOS app development.

(1) *Complete Migration*: #4 Apps initially developed in Objective-C gradually replaced it with Swift, and eventually used Swift only.
(2) *Swift Dominance from Start*: #77 Apps started with zero or a minor presence of Objective-C, and later transitioned entirely to Swift.
(3) *Swift Initiation with Objective-C Adoption*: One App started with pure Swift, and later introduced (and kept) a minor portion of Objective-C.
(4) *Swift Usage for New Components*: #30 Apps started with Objective-C and use Swift code for new components, but kept existing Objective-C code.
(5) *Gradual Shift to Swift*: #17 Apps show a steady decline in Objective-C usage as Swift was progressively integrated, and eventually dominated.
(6) *Stable Coexistence*: Two Apps maintain a consistent use of Objective-C alongside a predominant use of Swift throughout their development life cycle.

[9]https://wire.com/en
[10]https://www.privateinternetaccess.com
[11]https://github.com/fossasia/susi_iOS
[12]https://wordpress.org
[13]https://simplenote.com
[14]https://apps.apple.com/app/id1254612844

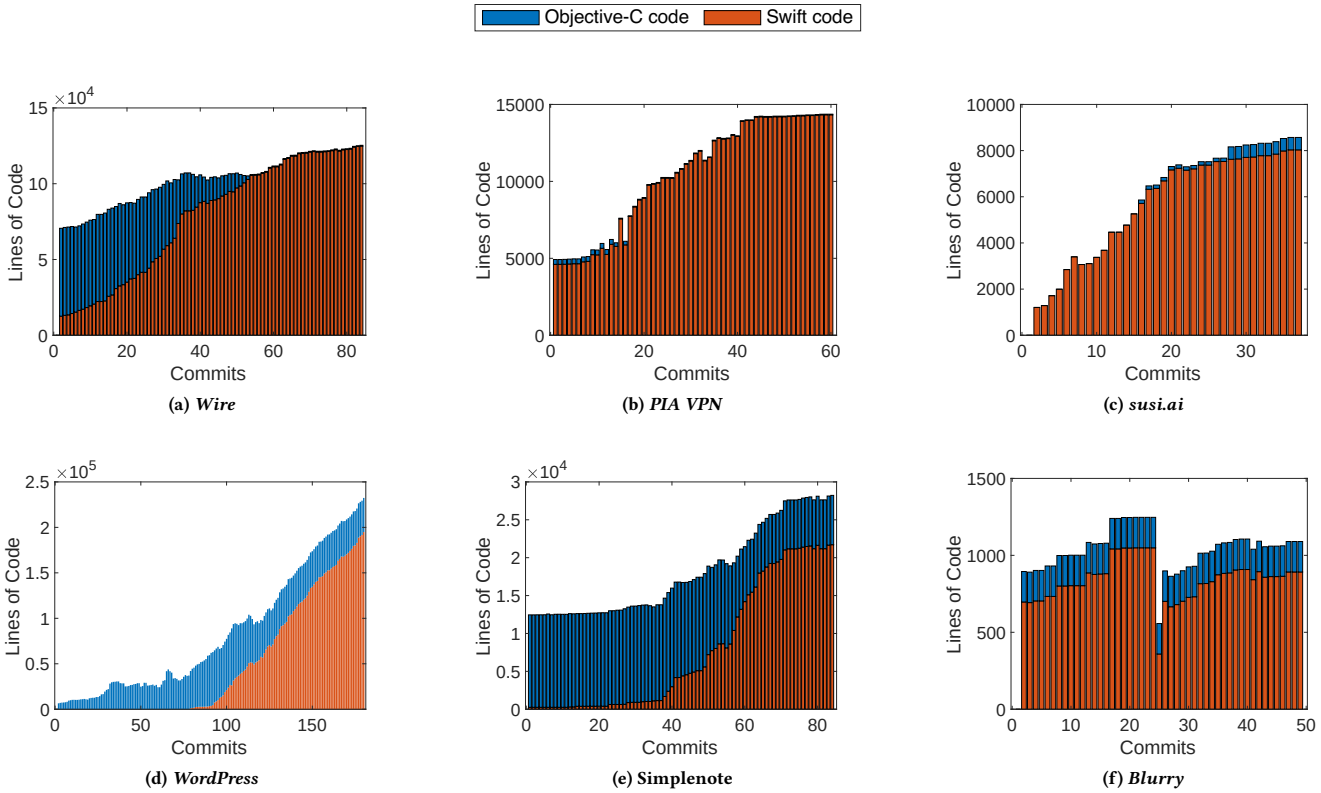


Figure 5: Language adoption patterns found in six exemplar apps.

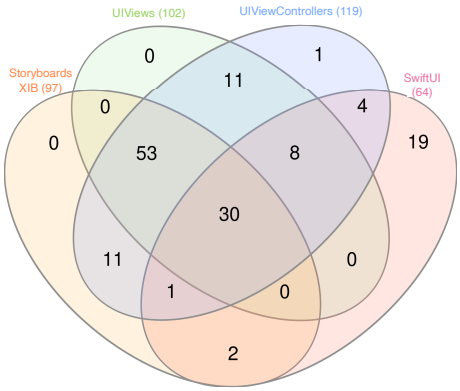


Figure 6: Distribution of the adoption of UI frameworks.

These patterns reflect the evolving landscape of iOS app development, with a marked shift towards Swift, either partially or completely, across most applications. It should be noted that the remaining 9 apps from the count above, reflected a consistent presence of Objective C code only.

Finding #3: Apps introducing Swift code exhibited six different evolutionary patterns, indicating, for example, the co-existence, with different proportions, of languages (for different purposes), the usage of Swift for new components, or cases of complete migration.

4.3 Adoption of UI frameworks (RQ_2)

Figure 6 shows the distribution of UIKit and SwiftUI adoption when we consider the last commit of each app. The figure identifies four main sets, each representing a technique for defining UIs.

- **Storyboard/XIB.** Apps that have at least two Storyboards or an XIB. The constraint on the minimum number of Storyboards is because the launch screen, the first screen shown when the app is launched, is usually specified as a standalone storyboard.
- **UIViews.** Apps with at least one UIKit View written in either Objective-C or Swift.
- **UIViewControllers.** Apps that have at least one UIKit ViewController written in either Objective-C or Swift.
- **SwiftUI.** Apps with at least one SwiftUI View.

The four groups highlight that 19 apps are only based on SwiftUI. All apps that have UIKit Views written as code also have one or more associated ViewControllers. This is because Views define the visual

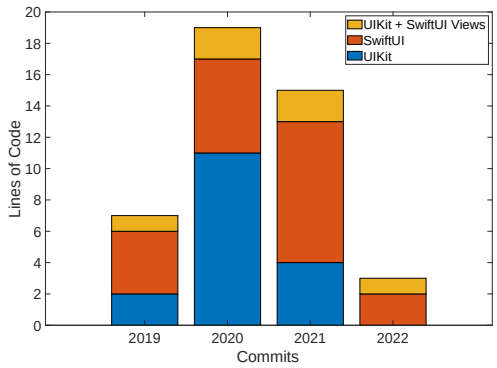


Figure 7: UI frameworks breakdown in new projects.

characteristics of GUI elements, yet they need a ViewController to control their behavior. 32 apps use both UIKit and SwiftUI at the same time. These apps historically adopted UIKit and moved to SwiftUI to develop new GUIs, or transformed some of the existing ones from UIKit to SwiftUI. 65 apps only use the classic UIKit: Views, ViewControllers, and their attributes are specified through a Storyboard or a XIB and managed through ViewControllers; 53 of them also have Views specified in their source code.

24 apps do not use Storyboards or XIBs and define UIKit Views and ViewControllers through code: 1 of them uses just ViewControllers to configure the associated default view programmatically, and 12 of them also embed SwiftUI code. While a Storyboard eases the definition of the layout of the different screens and the navigation between them, the use of pure code can be seen as a choice of homogeneity: There is only one way in which the app is defined, and this is fully visible in the same source files. It is also a way to avoid discrepancies between graphical specifications and their implementation.

In conclusion, most apps use just UIKit, and the vast majority mixes graphical specifications (i.e., Storyboards and XIBs) and code. 12 apps only use code and UIKit, while 19 adopt only SwiftUI.

Finding 4#: 19 apps have fully adopted SwiftUI, indicating a shift towards modern UI practices. Meanwhile, 32 apps merge UIKit with SwiftUI, showing a trend of incorporating SwiftUI into existing UIKit frameworks. Most of the apps (65) still rely on traditional UIKit methods using Storyboards or XIBs. A few apps (24) opt for a purely programmatic approach with UIKit, bypassing graphical specifications. This diversity highlights a gradual transition towards SwiftUI, while UIKit remains prevalent in app development.

4.4 Migration from UIKit to SwiftUI (RQ2)

To better explain the dynamics of the adoption of the new framework, We categorized the apps into four groups based on how and when they adopted SwiftUI.

21 apps were built completely since the beginning using SwiftUI (2 of them added Storyboard for minor purposes). The first commits of these apps appeared, of course, after June 2019, when SwiftUI was introduced and released. The modern features of SwiftUI motivated developers to rule out UIKit, which became (too) complex over time as new features were added, in favor of the new framework. These apps may contain a few UIKit elements wrapped into SwiftUI views, as SwiftUI has not reached the level of completeness of UIKit in terms of features.

As for the apps whose first commit dates after the announcement of SwiftUI in June 2019, note that 21 out of 44 apps adopted SwiftUI as the only UI framework being used since the beginning (2 of them complemented with a small amount of Storyboards). Figure 7 shows the distribution of UIKit/SwiftUI projects grouped by the year of their first commit. One can observe that SwiftUI has been increasingly chosen as a reference UI framework since its introduction. The figure highlights in yellow, the percentage of projects that adopt a mix of UIKit and SwiftUI components.

Finding #5: Among 140 analyzed apps, 64 projects adopted, at least partially, SwiftUI. The pace of migration to the new technology appears to be slower than what can be observed from Objective-C and Swift. However, new projects are mostly based on SwiftUI.

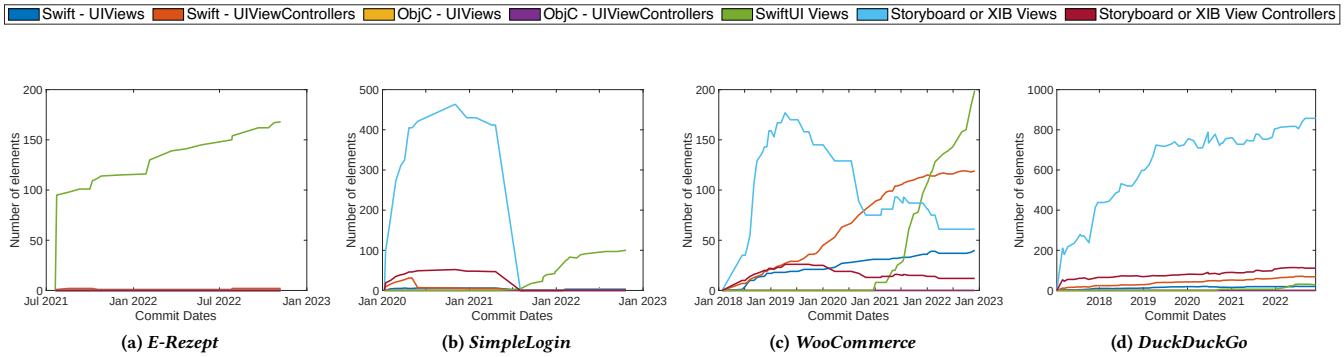
We then analyzed manually how single apps migrated to SwiftUI. Figure 8a shows the behavior of *E-Rezept*[15], a classic example of SwiftUI app. Since its first commit in July 2021, it has been characterized by a high number of SwiftUI views and a negligible number of UIViewControllers, encoded in Swift, used to integrate Share Sheet, only available as UIKit feature in a SwiftUI View.

Eight apps originally built with UIKit switched completely to SwiftUI, thus the developers rewrote all the GUI-related code. Although UIKit is still officially supported and is still an option when starting a new project, the "preferred" solution is now SwiftUI. Apps in this group decided to fully concentrate on adopting SwiftUI to address the present and be ready for the future. Figure 8b shows the evolution of *SimpleLogin*[16], which appeared in January 2020 as a UIKit app. It used Views and ViewControllers in Storyboards. SwiftUI replaced most of these elements starting in August 2021, witnessing a rapid replacement of UIKit elements with SwiftUI ones.

Five apps added a significant quantity of SwiftUI Views. All these apps introduced many new views specified in SwiftUI without reducing the number of UIKit-related Views. This category contains the most "hybrid" apps: at some point, they ceased the development of UIKit views to build them instead with SwiftUI. Figure 8c contains an example of this behavior. *WooCommerce*[17], a longstanding UIKit app, started introducing a significant quantity of SwiftUI in January 2021 with the addition of multiple views. This addition does not correspond to a reduction in the number of UIKit elements, but it corresponds to a stall in their growth.

30 apps make a small number of SwiftUI views co-exist with UIKit-related ones. The number of UIKit-related Views did not decrease, which means that SwiftUI Views were added to existing

[15]https://github.com/gematik/E-Rezept-App-iOS
[16]https://simplelogin.io/
[17]https://woo.com



(a) ***E-Rezept*** **(b)** ***SimpleLogin*** **(c)** ***WooCommerce*** **(d)** ***DuckDuckGo***

Figure 8: UI framework use in some relevant apps.

UIKit views instead of substituting them. Specifically, 13 apps out of 30 were forced to adopt SwiftUI, since it is the only solution for creating home screen widgets, a feature announced in June 2020 and released with iOS 14. Figure 8d shows the history behind *DuckDuckGo*[18], a quite complex app in terms of views. It made limited use of SwiftUI, starting from September 2020, to add home screen widgets and some custom SwiftUI views. The number of UIKit elements did not decrease; instead, it continued to grow.

Our analysis highlights four distinct patterns in adopting SwiftUI:

(1) *SwiftUI Exclusive*: 21 apps were built exclusively with SwiftUI from the beginning, thus reflecting a commitment to the modern features of the framework.
(2) *Complete Transition to SwiftUI*: Eight apps, which were originally built with UIKit, later completely transitioned to SwiftUI, thus indicating a strategic shift to future-proof their code base.
(3) *Hybrid Integration*: Five apps introduced a substantial number of SwiftUI elements while maintaining their UIKit components, showcasing a blend of the two frameworks.
(4) *Modest SwiftUI Addition*: 30 apps added a small number of SwiftUI views alongside existing UIKit elements, representing a cautious, incremental adoption of SwiftUI.

Finding #6: Similarly to what happened for programming languages, the evolution across different UI frameworks featured different (four) patterns, including the Exclusive use of SwiftUI, a complex transition toward SwiftUI, and a substantial or modest introduction of SwiftUI components.

5 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between theory and observation. Primarily, this threat concerns the extent to which the considered metrics (proportion of code written in a given language and of UI elements specific to a given UI framework) are suitable to address our research questions. In the end, the evolution of programming language proportions (Objective-C vs. Swift) may not necessarily reflect migration patterns, but, for example, the addition of new components. Concerning the reliability of the measurement, we partially rely on existing (widely used) tools such as `cloc` and `Sitrep`, and, on top of them, we have defined our heuristics (e.g., to identify UI elements), which we have carefully tested.

Threats to *internal validity* concern factors, internal to our study, that could affect our findings. This study is purely observational, and we do not make any cause-effect relationship claim. That is, the reasons for some of the observed patterns may depend on projects' decisions we did not capture. For that purpose, this (quantitative) study needs to be complemented with a qualitative one.

Threats to *conclusion validity* concern the relationship between the analyses being conducted and the study's outcome. As this is mainly an observational study, our conclusions are based on (i) the observation of some evolution patterns, e.g., related to the use of different programming languages, and (ii) descriptive statistics.

Threats to *external validity* concern the generalizability of our findings. While we have conducted our analysis on a set of 140 pretty diverse (in terms of domain) apps, further empirical evidence is needed, especially for closed-source apps.

6 RELATED WORK

The closest related research to ours concerns iOS applications and their quality measurement over time. Although mobile applications are usually less complex than desktop-class software, it is still vital to ensure high maintainability and reliability. For example, the adherence to object-oriented design patterns and clean code principles may produce benefits in performance and development times.

Habchi et al. [11] statically analyze the source code of 103 Objective-C and 176 Swift open-source apps to identify the presence of six different types of *code smells* that negatively affect the performance and maintainability of code: singleton abuse, massive view controller, heavy enter-background tasks, ignoring low-memory warnings, blocking the main thread, and download abuse. On the one hand, iOS apps exhibit the same percentage of code smells regardless of the chosen programming language. On the other hand, iOS apps seem to be less prone to code smells than their Android counterparts. The most frequent code smells are—similarly to what was found in conventional software [16]—lazy class, long method,

[18] https://duckduckgo.com

and message chain, while less than 18% of apps were affected by the massive view controller. By analyzing the evolution of *long method* code smells, Habchi et al. found that such smells were introduced—in 138 cases out of 158—when the method was created, again similarly to what happens in conventional software [18]. A comparison with Android apps shows that code smells in iOS applications also correspond to smaller classes, while in Android apps, they correspond to larger, more complex classes. Conversely, Zhou et al. [21] analyze 444,129 bug reports from 88 open-source desktop, Android, and iOS projects. Their study classifies bugs and evaluates how the obtained categories differ among platforms.

Domínguez-Álvarez et al. [7] provide an overview of the use of programming languages in the history of 25,231 third-party libraries and 32 open-source iOS native apps. Surprisingly, 95% of the libraries released before the mainstream release of Swift and 53% of the libraries released after such an event, do not adopt it, preferring instead Objective-C or even C and C++. This is likely because Objective-C libraries can be used by apps written in Swift and Objective-C, while Swift apps can only use libraries written in Swift. Swift adoption by apps has been quicker than adoption by libraries. The presence of C and C++ code in apps usually results from integrating old libraries in the source code. The work by Domínguez-Álvarez et al. also reports an analysis of the adoption of newer versions of Swift in libraries. While commonalities exist between our work and what was done by Domínguez-Álvarez et al. [7], the works also have substantial differences. Not only our work is wider (140 vs. 38 apps), but, also, while Domínguez-Álvarez et al. look at the programming language distribution, we also study evolutionary patterns (within apps' lifecycle) of programming languages over time, along with an evaluation of the adoption of UI frameworks.

Other research on programming languages includes the performance of Swift and Objective-C. Gut et al. [10] conclude that Swift is faster than Objective-C when executing sorting tasks, thanks to optimizations available in the Swift compiler. Qualitative comparisons highlight advancements in modern features of Swift and the requirement for fewer lines of code compared to Objective-C. Our work is complementary to performance-related work because performance [10, 17] and qualitative [8] comparisons between Objective-C and Swift provide good insights into the choice made by developers to transition from one language to the other.

Finally, concerning UI frameworks, while some studies on the differences between UIKit and SwiftUI have been carried out [9, 20], our analysis of their adoption is still beyond the state of the art since it offers an overview of the alternated adoption of either framework and under which specific conditions such adoption manifests. This is useful, for example, to understand whether adoption phases and connected decisions impact on quality attributes of resulting apps.

7 CONCLUSIONS

The paper reports an in-depth analysis of 140 open source iOS apps highlight significant shifts and transition patterns in iOS app development. It reveals that most analyzed applications have either completely transitioned to Swift or use it alongside Objective-C for new code, maintaining legacy code in Objective-C. While UIKit remains the foundation for most apps, SwiftUI has emerged as a modern and robust alternative, particularly for new applications, indicating a trend towards SwiftUI for new graphical user interfaces, even as existing UIKit-based screens remain unchanged.

We plan to further extend the analysis in different directions. For example, we would like to qualitatively investigate the rationale of programming language and UI migrations, and, also, understand the impact on the apps' design.

ACKNOWLEDGES

This work has been partially funded by the national funding for MUR-PRIN project EMELIOT (2020*W*3*A*5*FY*).

REFERENCES

[1] Eugene Belinski. 2022. Swift versions. https://swiftly.dev/swift-versions
[2] Hudson Borges and Marco Túlio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *J. Syst. Softw.* 146 (2018), 112–129.
[3] Mihai Carabas, Costin Carabas, Laura Gheorghe, Razvan Deaconescu, and Nicolae Tapus. 2016. Monitoring and auditing mobile operating systems. *Int. J. Space Based Situated Comput.* 6, 1 (2016), 54–63.
[4] Apple Developer. 2019. Introducing SwiftUI. https://developer.apple.com/tutorials/SwiftUI
[5] Apple Developer. 2022. App Store Review Guidelines. https://developer.apple.com/app-store/review/guidelines/
[6] Apple Developer. 2023. UIKit Documentation. https://developer.apple.com/documentation/uikit/
[7] Daniel Domínguez-Álvarez, Alessandra Gorla, and Juan Caballero. 2022. On the Usage of Programming Languages in the iOS Ecosystem. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 176–180.
[8] Cristian González García, Jordán Pascual Espada, Begoña Cristina Pelayo García Bustelo, and Juan Manuel Cueva Lovelle. 2015. Swift vs. Objective-C: A new programming language. *IJIMAI* 3, 3 (2015), 74–81.
[9] Ethan Gilchrist. 2021. SwiftUI vs UIKit: A Case Study on How a Declarative Framework Can Improve Learnability of UI Programming. (2021).
[10] Kamil Gut, Maria Skublewska-Paszkowska, Edyta Łukasik, and Jakub Smołka. 2017. Comparison of Programming Languages on the iOS Platform in terms of performance. *Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska* 7, 3 (Sep. 2017), 33–36.
[11] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. 2017. Code Smells in iOS Apps: How Do They Compare to Android?. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 110–121.
[12] Firdaus Bin Harun. 2019. Review of iOS Architectural Pattern for Testability, Modifiability, and Performance Quality. *Journal of Theoretical and Applied Information Technology* 97, 15 (2019).
[13] Hansen Hsu. 2017. A short history of Objective-C. *CORE* (2017), 62–65. http://s3.computerhistory.org/core/core-2017.pdf
[14] Paul Hudson. 2023. What's new in Swift? https://www.hackingwithswift.com/swift
[15] Apple Inc. 2023. swift/CHANGELOG.md - GitHub. https://github.com/apple/swift/blob/main/CHANGELOG.md
[16] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empir. Softw. Eng.* 23, 3 (2018), 1188–1221.
[17] Harwinder Singh. 2016. Speed performance between swift and objective-C. *Int. J. Eng. Appl. Sci. Technol* 1, 10 (2016), 185–189.
[18] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 403–414.
[19] Steffen Vaupel, Gabriele Taentzer, René Gerlach, and Michael Guckert. 2018. Model-driven development of mobile applications for Android and iOS supporting role-based app variability. *Softw. Syst. Model.* 17, 1 (2018), 35–63.
[20] Piotr Wiertel and Maria Skublewska-Paszkowska. 2021. Comparative analysis of UIKit and SwiftUI frameworks in iOS system. *Journal of Computer Sciences Institute* 20 (2021), 170–174.
[21] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. 2015. A Cross-Platform Analysis of Bugs and Bug-Fixing in Open Source Projects: Desktop vs. Android vs. IOS. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering* (Nanjing, China) *(EASE '15)*. ACM, New York, NY, USA, Article 7, 10 pages.