Online-Journals.org

iJIM International Journal of Interactive Mobile Technologies

iJIM | eISSN: 1865-7923 | Vol. 17 No. 23 (2023) | OPEN ACCESS

https://doi.org/10.3991/ijim.v17i23.43829

PAPER

Porting a Native Android App to iOS: Porting Process Shown by the Example of the "Schoolstart Screening App"

Mattias Rauter, Josef Wachtler, Martin Ebner(✉)

Educational Technology, Graz University of Technology, Graz, Austria

martin.ebner@tugraz.at

ABSTRACT

The two mobile operating systems Android and i(Pad)OS have dominated the smartphone and tablet market for years and app providers have to offer their apps for both systems in most cases in order to be competitive or to be able to reach the majority of potential customers. In native app development, separate applications have to be written and maintained for each platform. Often, apps are developed for one platform first and the second app is developed at a later stage, after some feedback could be collected. This porting from one system to the other can be either (partially) automated or manual, but in any case, it has its challenges. Both systems were designed with different approaches and differ greatly in some parts from each other – not only visually, but also in terms of the underlying structure. To illustrate the porting process, the Android app "Schoolstart Screening App", which was developed for the Federal Ministry of Education, Science and Research of Austria by Graz University of Technology at the OU Educational Technology, was ported so that it can be used also on iPads. Automated approaches were discussed and the chosen process is explained to get a good overview of the topic.

KEYWORDS

Android, iOS, native apps, porting automatically, porting manually

1 INTRODUCTION

Mobile computing has become increasingly important in recent years. In the beginning, several companies were represented on the mobile operating system market, but now two major players remain with a market share of 99.39% (end of 2022, [1]): *Android* and *i(Pad)OS*. To be able to offer one's own services in the form of an app to almost all potential users – the broad mass of Android users (71.75%, end of 2022) on the one hand and the significantly more well-paying iOS users [2] on the other – these two platforms have to be served. In 2019, the Federal Ministry of Education, Science and Research of Austria was faced with the challenge of porting the native Android app [3] "Schoolstart Screening App", which is intended to standardize and digitalize the elementary school enrollment process in Austria [4], to iOS. The technical development

Rauter, M., Wachtler, J., Ebner, M. (2023). Porting a Native Android App to iOS: Porting Process Shown by the Example of the "Schoolstart Screening App". *International Journal of Interactive Mobile Technologies (iJIM)*, 17(23), pp. 20–31. https://doi.org/10.3991/ijim.v17i23.43829

Article submitted 2023-08-07. Revision uploaded 2023-10-15. Final acceptance 2023-10-18.

was carried out by Graz University of Technology at the OU Educational Technology. In this paper, the process of porting the existing Android app to the iOS platform is discussed and the different possibilities – from automated to manual porting – are discussed. The potential difficulties are also explained in order to create a better understanding of the topic and the chosen approach of porting the app is shown.

2 RELATED WORK

There is a great desire, especially from the business world, to port native applications to other platforms in an automated way in order to save time and money. That is why there are different approaches to how this could be achieved. A few of them are presented here.

2.1 Porty

Xiaochao Fan and Kenny Wong focused their work on porting the user interface of Android applications to iOS [9]. They have developed a tool called *Porty* for this purpose. The whole source code and all other resource files have to be available and given to the tool. With the help of a specially adapted version of *LYCIA* [10], the layout of the Android app is parsed and transformed into different formats. *J2ObjC* [11] is used to translate the Java code of the Android app into Objective-C code for the iOS app. However, there may be limitations here, especially if platform-specific functions are used. In addition to the UI, event handlers are also transferred in order to be able to react to user input in the form of taps or swipe gestures. The names of the corresponding elements are translated using a specially developed *SymbolMappings* library. The tool can be used to transfer XML-based layouts – not programmatically defined user interfaces in Java – into programmatic Objective-C user interface construction code. This approach can also only be used to a limited extent, as many of the current Android apps are written in Kotlin and no longer in Java. The development of iOS apps using Swift instead of Objective-C is now also strongly recommended.

2.2 Kotlift

At the beginning of 2016, Valentin Slawicek published the project *Kotlift* on GitHub [5], which should automatically convert Kotlin code into Swift code. The focus was on a selected list of Kotlin-specific features and syntax that were supported, and all others were ignored for the moment. A full list of supported features is available in the project's README.md file. This considerable limitation means that not all projects written in Kotlin can be ported automatically with this tool. Another massive limitation are the supported versions. Kotlin v1.0.1 code can be converted to Swift v2.2 code. Since Kotlin v1.8.21 and Swift v5.8 are currently available and the languages are also developing very quickly, the tool is hardly relevant for current projects. *Kotlift* has now also been discontinued by the developer, so it is definitely not recommended to use it anymore.

3 PORTING ANDROID APPS TO IOS PLATFORM

In order to serve the majority of potential users of mobile devices, applications must be available on at least iOS and Android. Often it is also a requirement of the

client to make the app available for both platforms. It can be advisable to start with one platform at the beginning of the development and to concentrate fully on this. This means that adjustments can be made more quickly during development process if feedback has been collected with the first test versions, as only one and not several apps need to be changed. The second app can be rebuilt more easily afterward, when the first app has been found to be good by the market, clients, or customers. This process, i.e. porting an app from one platform to another, holds some challenges and can, under certain conditions, be (partially) automated or completely manual, as will be discussed below.

3.1 Challenges

The two operating systems Android and iOS were designed with different approaches, which is also reflected in many aspects of the systems. Android, which is based on the BSD Unix kernel, has been very open from the beginning, while Apple has tried to keep as much as possible under its control in iOS in order to have as much influence as possible on the user experience, e.g. when it comes to background activities that have a significant impact on the battery life of the devices and thus the user experience with Apple's hardware. Even though the two systems have become increasingly similar in recent years, there are still some fundamental differences that must be considered when porting and that make automatic porting difficult.

Access Rights Management is handled differently in Android and iOS. While Android followed an "all-or-nothing" approach for a long time, in which all necessary permissions for camera, location, photos, contacts, etc. were requested when the app was installed and the user could only decide for or against all permissions, iOS offered very early on to approve individual permissions during runtime or to adjust them later in the settings. It was therefore possible to use an app without having to share the exact location, even if the app would like to have it. This results in situations that have to be handled separately in iOS apps. Since full authorization could be expected during runtime in Android apps, this was not the case in iOS and additional code has to be written for this in order to intercept this case. In the meantime, however, Android has switched to a cause-related rights management, but the type and granularity of the authorization queries still varies.

Background Tasks can be used in Android nearly without any restrictions, while Apple only allows defined activities in the background. When (automatically) porting apps with heavy background activity usage, it might be very difficult or even impossible, to implement the same behavior in iOS.

Look&Feel of mobile applications is very much shaped by the respective companies. In order to be able to publish apps via Apple's App Store, the applications should comply with the "Human Interface Guidelines" [6]. These specify how the apps should behave, which animations should be used, how long they should last, which icons should be used, how they should be positioned and what spacing and padding should be used. Fonts, font sizes and colors are also defined. All this serves to ensure a consistent experience across all of Apple's operating systems, like iOS, macOS, tvOS and some more. Google uses Material UI [7] for Android apps to achieve a uniform appearance.

Back Button is always available on Android, either in the form of a physical button or as an on-screen software button or via a special swipe gesture at the bottom of the screen, as shown in Figure 1. In addition to this, Android apps can also have a back button at the top left of the screen, as is known from many iOS apps. iOS, on the other hand, does not have a system-wide back function, but its own patterns

for implementing different types of "steps back". These are usually dependent on the way the current window was opened and which logical operation the "go back" performs, whether, for example, the currently displayed information is closed or a step is taken back within the process. Some possibilities are shown in Figure 2. The lack of an always-available go-back option, as is the case on Android, makes porting to iOS extremely difficult, as explicit ways must be created to implement the go-back logic. A simple approach would be to automatically insert a back button on every screen, but that would again strongly influence the current user interface and possibly destroy it. A fully automatic approach is difficult to realize.

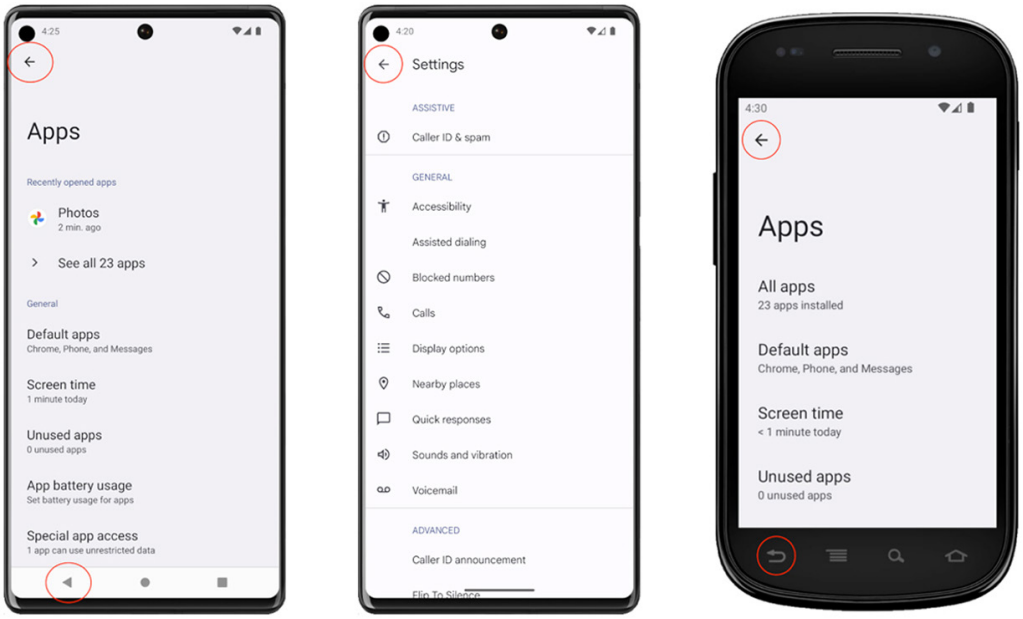

Fig. 1. Different types of "back buttons" on Android [8]

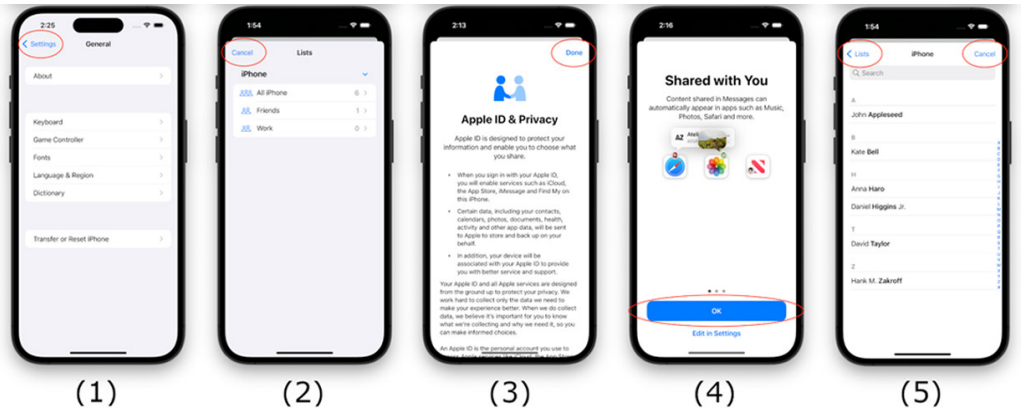

Fig. 2. Different types of "back-logic" on iOS [8]

Floating Action Button was introduced in Material Design by Google in 2014 [22] and is a very prominently placed primary-action button. Different versions of it are shown in Figure 3. Even though it is technically possible to recreate the button in iOS, it still feels unfamiliar in most iOS applications. Apple provides other design patterns for this type of primary action button, as shown for example in Figure 4. However, since other buttons may also be placed at these positions in Android, an automatic conversion may be difficult.

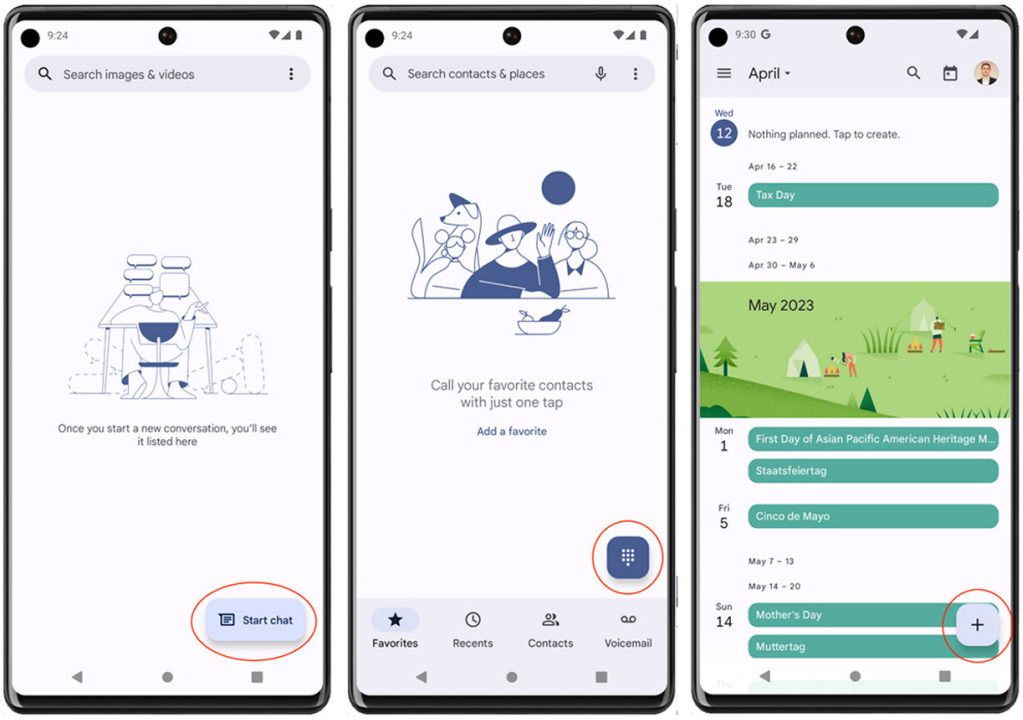

Fig. 3. Examples of Floating Action Button (FAB) on Android [8]

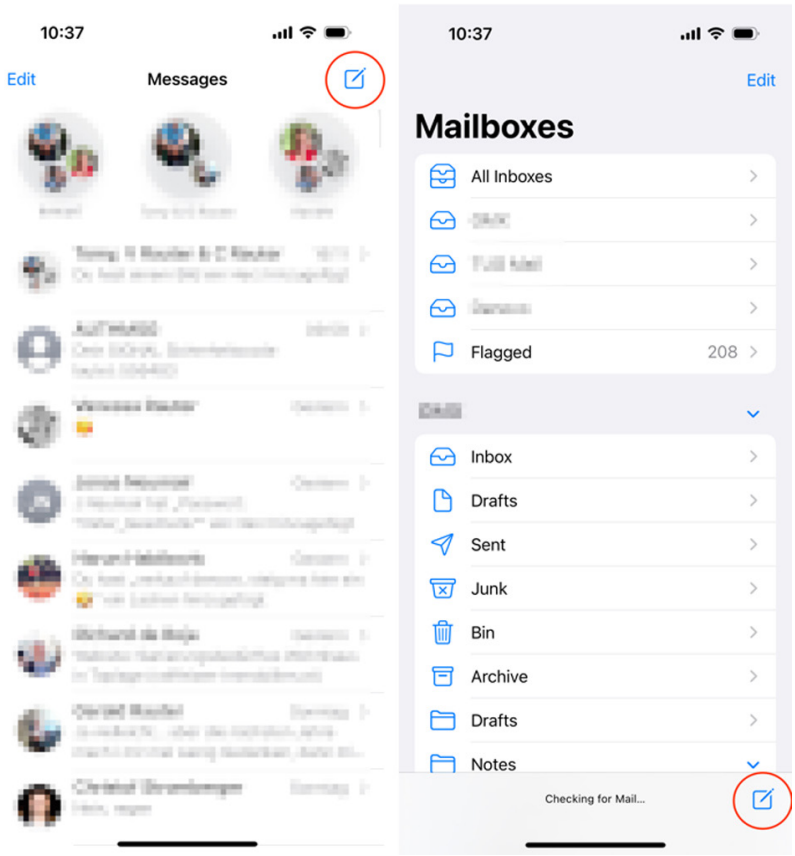

Fig. 4. Examples of primary action buttons on iOS [8]

3.2 Automatic porting

Automated approaches to porting Android applications to iOS have been very limited so far, yet there are some ways in which this could be made possible. The extent to which they were useful in the present case of the "Schoolstart Screening App" is discussed below.

Porty [9] could not be used for this project because it was written in Kotlin, which is not supported. In addition, the Android application was not yet 100% finished and the deadline was very tight, i.e. after completion of the Android app there was not much time left for porting and possible reworking.

Kotlift [5] could not be used due to the very outdated supported Kotlin and Swift versions. The limited supported functionality would not have been sufficient even if the versions had been compatible.

Kotlin/Native [12] offers the possibility to compile Kotlin code into binaries in order to be able to use them on other systems. The following platforms are supported [13]:

1. macOS
2. iOS, tvOS, watchOS
3. Linux
4. Windows (MinGW)
5. Android NDK

Kotlin/Native eliminates the need for the Java Virtual Machine to run the Kotlin code and the business logic of the Android app can be transferred to iOS without having to re-implement it. This partially automated porting process can definitely be advantageous for very logic-intensive applications, but for user interface-heavy apps, the added value is very limited or the disadvantages could outweigh the benefits. The debugging effort is increased enormously, since parts of the iOS app are no longer available as Swift or Objective-C code, but as binary, which can be addressed or called, but not easily analyzed. In addition, for an extension of the encapsulated logic, Kotlin must be mastered, which in the case of this project was only available to a limited extent in the iOS team. Another complicating factor was that there was too little time between the final Android version and the planned completion date of the iOS app to do the necessary integration work of the binary into the iOS app.

3.3 Manual porting

Manual porting is the separate re-implementation of existing applications for another platform. In the case of the "Schoolstart Screening App", three phases were necessary. First, the current status had to be analyzed to get an overview of the system landscape and the applications. Then, requirements could be derived from the findings, which were implemented in the third phase.

Analysis of the current application and system environment was done by reading all available project-relevant documents, especially regarding the technical implementation of the current system. In meetings with stakeholders of the project, a general overview was provided and open questions were clarified as quickly and efficiently as possible. Access to the current test system was also provided so that the latest software could be tested directly. The schematic illustration of the set up

architecture is shown in Figure 5. A web frontend enables simple data management and the backend offers all the necessary functions for operating the app via a REST API and shields the data from the outside. As practical tests had already been carried out with the Android app in advance, some realistic test data was also available to facilitate the process. Every step in the app could be carefully examined in order to be able to precisely identify implicit functions.

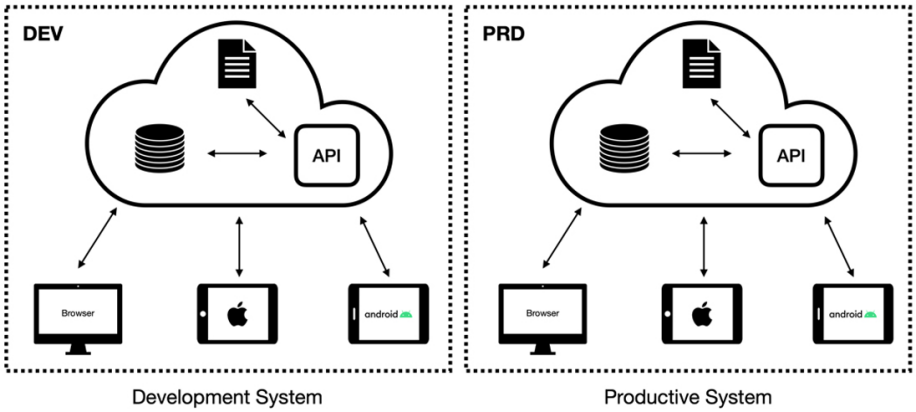

Fig. 5. System architecture of the "Schoolstart Screening App" system [8]

Requirements engineering is carried out on the basis of the previously analyzed current status of the existing system. All necessary steps for the fulfillment of the task must be specified and were defined in the form of *user stories*. For this purpose, the desired behavior is written down in an easily understandable way and enriched with *acceptance criteria* that concretize the behavior. It is not sufficient to repeat the same steps as for the Android app, as the goal must be to firstly implement the iOS app as efficiently as possible, i.e. to avoid any mistakes or uncertainties in the specification of the Android app and not to repeat them. This is especially important because the Android development has been running for a long time and there have also been several feedback cycles that have resulted in one or another adjustment in the app. And secondly, iOS-specific characteristics must be taken into account. It is particularly important that the iOS app to be developed complies with the paradigms and guidelines required by Apple so that it can later be published in the Apple App Store. Particularly regarding design, some adjustments had to be made to meet the requirements. Based on the user stories and the defined schedule, a roadmap was created as shown in Figure 6.

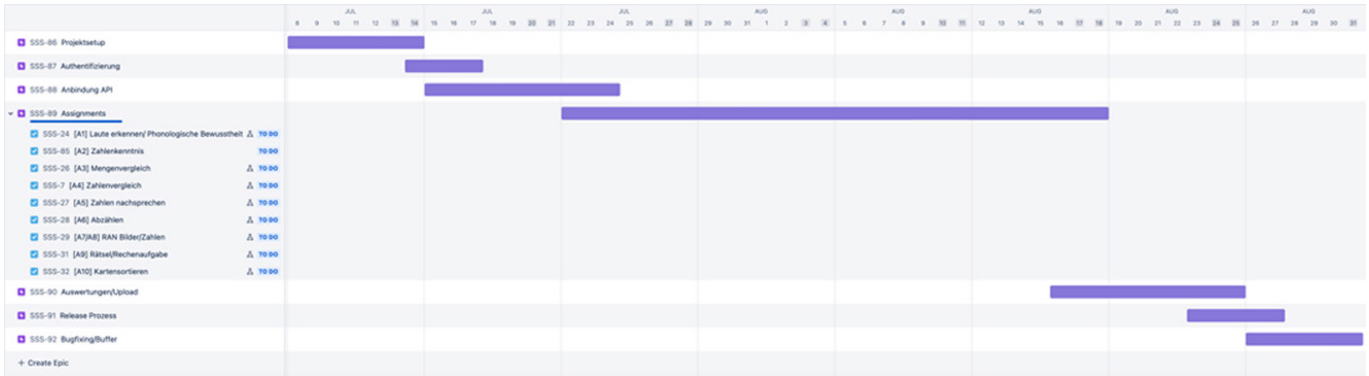

Fig. 6. Roadmap of the iOS implementation [8]

Implementation phase started with setting up a GIT repository and checking in a new XCode project. The use of feature branches simplified the work enormously, especially if the server application is also extended parallel to the development of the iOS app. Since the teams have agreed to allow breaking changes while the system is not yet live, the deployment of different API versions on separate environments could be coordinated and different branches could be used to build separate apps for different environments. External dependencies were managed via *CocoaPods* [14], which assisted with version management of external frameworks. During the development phase, the Kanban board, a virtual board with all tickets of the project on it, visualizing the current status – *Todo*, *In Progress* and *Done* – provided a constant overview of the project's progress and was updated regularly.

4 RESULTS

The automatic porting approaches would not have led to the desired result, so in the case of the "Schoolstart Screening App", manual porting was chosen. Through this approach it was possible to create almost identical conditions for the users of the apps on both systems and still make the apps feel familiar on their own operating systems, as Figures 7–11 show. This is particularly clear when using the system's own alerts, as seen in Figure 12.

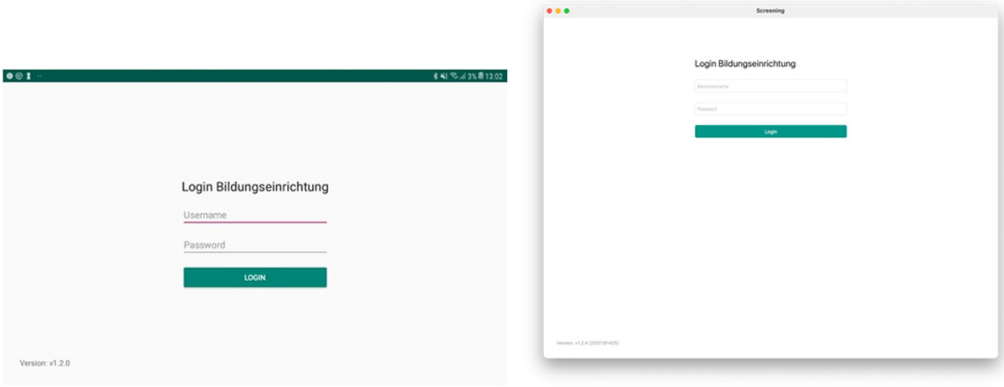

Fig. 7. Comparison of the "Login" screen on Android (left) and iOS (right)

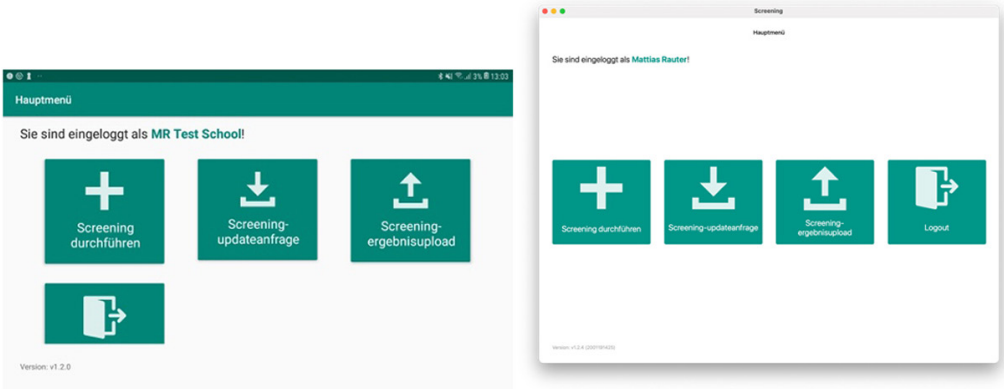

Fig. 8. Main menu of the "Schoolstart Screening App" on Android (left) and iOS (right)

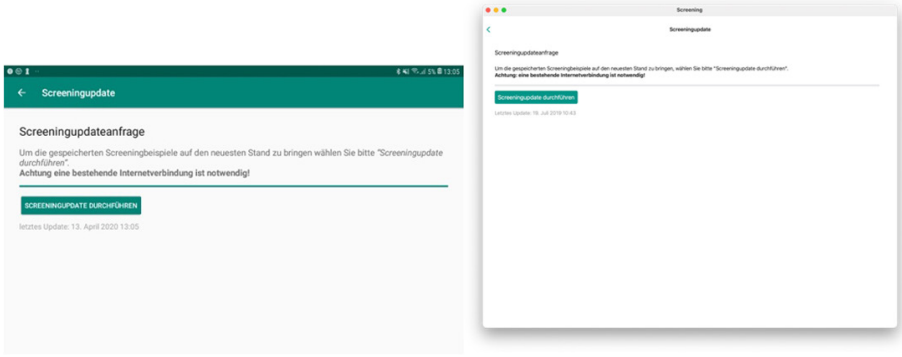

Fig. 9. Submenu UI differences for Android (left) and iOS (right) apps

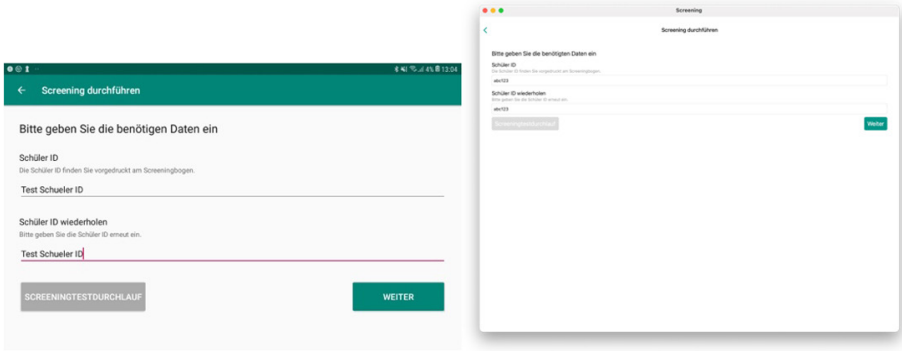

Fig. 10. Input of user data of the different systems (Android left, iOS right)

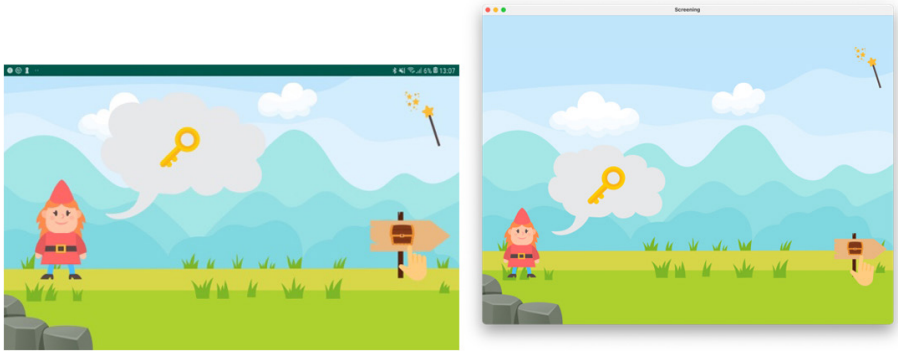

Fig. 11. Comparison of the very similar assessment visualization on Android (left) and iOS (right)

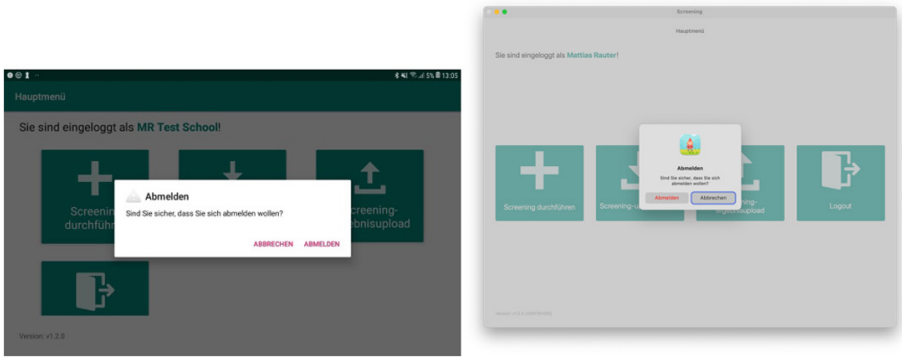

Fig. 12. Different specific alert views provided by Android (left) and iOS (right) are used that the apps feel more natural within the platform

5 FUTURE WORK

The increasing similarity between the two dominant mobile operating systems, Android and iOS, and the enormous progress made in the field of *artificial intelligence (AI)*, especially in recent times, give reason to hope that fully automated porting will become more likely in the near future. Many apps are available for Android and iOS and could serve as training data for AI tools like *chatGPT* [15], Microsoft's *GitHub Copilot* [16] or similar to achieve better results soon. With almost 5 million apps in the Google (*Play Store*) and Apple (*App Store*) app stores alone by the end of 2022 [17][18], the market for mobile applications is very large and, as discussed earlier, their presence on both operating systems is very often commercially viable, which could accelerate research in this direction due to great commercial interest.

Another approach would be to develop applications for both platforms with cross-platform frameworks. In this case, a code base is written, which can be extended by individual native code on both platforms, and then compiled for both systems. Porting is then only necessary to a very limited extent or not at all. Two very widespread frameworks [21] for this are *Flutter* [19] and *ReactNative* [20]. Both approaches have many advantages, but also some disadvantages that need to be looked at in more detail in another paper. Furthermore, the use of these or similar frameworks does not solve all the challenges identified in this paper, such as dealing with access rights or background activities.

6 CONCLUSION

Porting mobile applications from Android to iOS is still a very complex process, especially since the two operating systems are clearly designed differently, despite a functional convergence in recent years and the latest versions. Many decisions taken by Apple and Google in the design of the two systems make the automated porting of applications difficult. In order to be able to offer high-quality apps, the only way left is usually manual porting and thus a de facto new, independent app version for iOS devices. Automated approaches can help in some cases, but so far, they are not good enough to fully work out in more complex cases. A selection of reasons why manual porting is preferred to automatically porting is given in Table 1.

Table 1. Reasons for choosing a manual porting approach over an automatic one [8]

Manual Porting	Automatic Porting
The newest state-of-the-art frameworks and versions can be used (e.g. newest features, highest security, future-proof…)	Limitations regarding the used source and destination versions based on chosen porting tool
Clear and structured project management based on accurate estimations of necessary tasks	Very hard to estimate the effort needed to fix automatically generated semi-working code and plan a reliable roadmap
iOS specific functions and patterns can be used to provide readable and maintainable source code	The "Android-way" of implementing features and Android specific patterns are carried over to iOS
UI/UX guidelines for iOS can be implemented easily	Android guidelines are transferred to iOS, which can lead to issues e.g. regarding missing back button logic etc.
No dependencies on third-party providers' porting tools	Dependent on others in case of bugs or issues with the software and for further development

7 REFERENCES

[1] Global mobile OS market share 2022 – Statista. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/. [Accessed: July 28, 2023].

[2] L Rabe. Apps – Umsatz nach App Store weltweit 2021 – Statista. Available: https://de.statista.com/statistik/daten/studie/802760/umfrage/schaetzung-des-umsatzes-mit-apps-nach-app-store-weltweit/. [Accessed: July 29, 2023].

[3] Paul Krassnig, *School Start Screening Tool.* 2021.

[4] Schulanmeldung (Einschreibung). Available: https://www.oesterreich.gv.at/themen/bildung_und_neue_medien/schule/4/Seite.110031.html. [Accessed: July 29, 2023].

[5] studoverse/Kotlift: Kotlift is the first source-to-source language transpiler from Kotlin to Swift. Available: https://github.com/studoverse/Kotlift. [Accessed: July 27, 2023].

[6] Apple Inc. Human Interface Guidelines – Human Interface Guidelines – Design – Apple Developer. Available: https://developer.apple.com/design/human-interface-guidelines/guidelines/overview/. [Accessed: July 27, 2023].

[7] Google. Material Design. Available: https://m3.material.io/. [Accessed: July 29, 2023].

[8] Mattias Rauter, *Portability of Mobile Applications*, 2023.

[9] X. Fan and K. Wong, "Migrating user interfaces in native mobile applications: Android to iOS," in *Proceedings – International Conference on Mobile Software Engineering and Systems*, MOBILESoft 2016, 2016, pp. 210–213. https://doi.org/10.1145/2897073.2897101

[10] LYCIA. Available: https://code.google.com/archive/p/lycia/. [Accessed: July 28, 2023].

[11] J2ObjC – Google for Developers. Available: https://developers.google.com/j2objc. [Accessed: July 30, 2023].

[12] Kotlin Native – Kotlin Documentation. Available: https://kotlinlang.org/docs/native-overview.html. [Accessed: July 25, 2023].

[13] Kotlin Native – Kotlin Documentation. Available: https://kotlinlang.org/docs/native-overview.html#target-platforms. [Accessed: July 21, 2023].

[14] CocoaPods.org. Available: https://cocoapods.org/. [Accessed: July 21, 2023].

[15] Introducing ChatGPT. Available: https://openai.com/blog/chatgpt. [Accessed: July 27, 2023].

[16] GitHub Copilot · Your AI pair programmer. Available: https://github.com/features/copilot. [Accessed: July 28, 2023].

[17] L Ceci. Google Play Store: number of apps 2023 – Statista. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Accessed: July 29, 2023].

[18] David Curry. App Store Data (2023) – Business of Apps. Available: https://www.businessofapps.com/data/app-stores/. [Accessed: July 29, 2023].

[19] Flutter – Build apps for any screen. Available: https://flutter.dev/. [Accessed: July 30, 2023].

[20] React Native · Learn once, write anywhere. Available: https://reactnative.dev/. [Accessed: July 29, 2023].

[21] Cross-platform mobile frameworks used by global developers 2022 – Statista. Available: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. [Accessed: July 30, 2023].

[22] Steve Jones, *User Experience Implications of the Floating Action Button*, 2016.

8 AUTHORS

Mattias Rauter is a Computer Science master student at Graz University of Technology and currently working as managing director at Denovo, a software development and consulting company in Graz (E-mail: rauter@denovo.at).

Josef Wachtler is currently working at the Department of Educational Technology at Graz University of Technology as an Edtech-developer. Furthermore, he holds a PhD in computer science and assists in supervising Master's and Bachelor's theses. His research-interests are in the field of video-based learning used in different settings like schools, universities or MOOCs (E-mail: josef.wachtler@tugraz.at).

Martin Ebner is currently Head of the Department Educational Technology at Graz University of Technology and therefore responsible for all university-wide e-learning activities as well as a Senior researcher for educational technology (E-mail: mebner@gmx.at).