Automatic Code Generation for a Cross-Platform Indoor Navigation App

DORIAN DROST
REBECCA FORTMANN
CHRISTIAN KULLIK
DUSTIN MATZEL
BETTINA REGLIN

Introduction

In this chapter we will discuss the initial situation and the problem formulation of our project. In chapter two we will then discuss possible solutions towards this problem, before we explain the procedure of our crosscompilation in chapter three. In chapter four we will discuss the problem of map representation in iOS in more detail. At last, in chapter five we present our results and discuss them with regard to future work in chapter six.

The UniMaps Application

UniMaps is an Android Application that allows barrier-free navigation on the campus of the University of Bielefeld. First developed as a students' project within the course "Techniken der Projektentwicklung" (Software Engineering), it is developed further since then by students on behalf of the ZAB (Zentrale Anlaufstelle Barrierefrei, Central Department for Accessibility). Beneath navigating, the application focuses on supporting a diverse group of users - namely handicapped and international students - and provides several important features as displaying the canteen menu and the tram schedule and providing information of the PEVZ (Personen- und Einrichtungsverzeichnis, Directory of Staff and Departments).

To reach an extend group of users, it is necessary to make UniMaps available not only on Android, but also on iOS-devices.

Multi-platform Applications

Smartphone applications nowadays mainly run on one of two operating systems, that are Android and iOS. Those two operating systems, of course, make different demands towards the process of development. While there are decent differences in the structure of the projects, the main difference lies in the programming language: Android applications are written in Java or Kotlin Code, iOS applications demand Swift or Objective-C code.

On the other hand, when having the same application for Android and iOS, most parts of both applications are identical of course. When thinking of the UniMaps application, it is obvious, that both the Android and the iOS application would need an implementation of the A*-Algorithm, a parser for the Canteen and PEVZ APIs, the implementation of the tram schedule, etc. The differences mainly lie in the presentation of the user interface, the technical background regarding the database and the overall project structure.

Possible solutions

After we now have an overview of the UniMaps Application, we will discuss the methods that are common when developing multi-platform applications in this chapter. We will then evaluate those methods with regard to our needs and choose the one that fits our project best.

Initial situation

At the beginning of this project, we have a working Android application written in Java Code. This code follows the structure of object-oriented programming and is written natively to fit the frameworks and structures used in an Android application.

The goal we aim at is to have a duplicate of the application that runs on iOS devices. While both applications should provide the same features, we expect the design and usability of both to be different, depending on the common standards of Android or iOS applications. Creating this duplicate should, of course, be as little time-intensive as possible. We also aim at both applications following a shared structure, that allows to easily apply changes made in one application to the other.

Common standards in multiplatform applications

When developing applications for both Android and iOS smartphones, two main approaches exist that save one from programming the very same app twice, that are code generation and crosscompilation.

When using code generation, one code-basis is programmed in a higher level of abstraction. This code basis and the language it is written in is inherentely designed in order to be compiled in both Java Code (Android) and Objective-C Code (iOS). This compilation is, of course, not only limited to Java and Objective-C Code. Several frameworks also allow to create web-applications or applications for other systems. Xamarin[1] is an example of such a language.

Crosscompilation on the other hand, compiles existing code from one programming language into another (from Java Code into Objective-C Code or vice versa).

Requirements

Based on the current status of the UniMaps project, some demands arise towards the method to use. At first, the main goal of the method should be to minimize the amount of work that has to be done. This of course includes both the effort of using the method itself, such as the

1 https://dotnet.microsoft.com/apps/xamarin

preparation of the project. In particular, it would be beneficial for us, to reuse the code we already have, instead of re-writing it again in another language or framework.

A further, although less important, point is, that using and gaining know-how of the method itself also causes effort and costs man-hours. That also includes learning new programming languages or frameworks.

Comparison of different methods

We will now compare the methods of code generation and crosscompilation regarding the demands we formulated before.

The main benefit of code generation is the parallel development of applications for several systems. When writing code in, say, Xamarin one develops applications for several platforms that are of the same status at all time. In particular, code generation scales very well when serving additional platforms (i.e. when developing for Android and iOS, it is only little more effort to add a web-application). The approach of code generation also has disadvantages of course. The demand to program code in one language for several platforms can lead to difficulties regarding the way different applications are used usually. On different platforms, different ways of interacting with the user are common standard, that exclude each other to some extend. For example, when designing iOS applications, it is necessary to always provide a button that allows the user to go back onto the previous screen. On android, this button already exists in the lower bar of the screen, so it is not necessary in the application’s screen itself.

The main advantage of crosscompilation, on the other hand, is the ability to re-use code that already exists: When having an Android application written in Java, this code can be crosscompiled to Objective-C to get an iOS application. However, the Java code of the Android application is not inherently designed to be crosscompiled into other languages. This can lead to problems or difficulties while crosscompiling. Furthermore, when adding additional platforms, for every platform a cross-compilation pipeline has to be build, causing a higher complexitiy of the overall project.

Choosing the best method

Based on the current status of the project, the choice of the method is obvious: As we already have an existing Android project, we want to re-use this code, instead of re-writing it again as it would be necessary to perform the code generation approach. We therefore decide on crosscompilation as the method of choice. Furthermore, as we want our application to run only on Android and iOS, and not on further platforms, we would take no use of code generation’s greatest advantage at all.

Procedure of Code Generation

Refactoring

With the J2ObjC crosscompiler, it is possible to crosscompile Java code only without Android-specific components. This is obvious of course: As Android and iOS applications demand a different structure of the code, there is no direct equivalent to Android-specific parts such as Fragments or Activities. In particular, as the Java code is crosscompiled class-wise, a class we want to crosscompile, must not contain any Android-specific code.

As our project followed an object-oriented structure, the java parts and the Android-specific parts were not seperated well. To prepare the crosscompilation, we therefore had to refactor our project into a frontend and a backend. The frontend consists of all code that is Android-specific in some way (i.e. mainly the fragments and activities). The backend on the other hand, consists of Java code only, that has no dependencies to the platform it runs on. This includes the A*-algorithm, the canteen parser and the tram schedule, to name just a few. The whole backend can then be crosscompiled into Objective-C code. This refactoring demanded quite a high portion of the work done in this project.

Once the refactoring was finished, the following packages were separated in such a way, that they were crosscompileable:

- alias
- canteen
- database
- mainactivity (interfaces)
- navigation (including description generator)
- obstacles
- pevz
- searchhistory
- tramschedule
- userinputmapper
- utility

Crosscompilation

With the J2ObjC Crosscompiler we crosscompiled the backend from Java into Objective-C code. This crosscompilation is a stage of the CI-pipeline as described in the following chapter. Additionally, the crosscompilation is done every time the iOS project is built. To this end the Android project is added to the iOS project as a git submodule. We decided to use git submodules instead of an own repository for the backend. This saved us from maintaining a third repository. Furthermore, if we extracted the backend into an own repository, this repository would have had to be included in both the Android and the iOS project. With the submodule, the backend is still part of the Android repository and has to be included into the iOS project only.

The crosscompilation is done almost automatically. Only the bridging header that contains the names of all crosscompiled classes has to be updated manually whenever a new class is added to the crosscompilation pipeline. This bridging header is necessary for mixed-language applications (i.e. applications containing both Swift and Objective-C code). If the iOS application was written entirely in Objective-C, this bridging header would not be necessary and the crosscompilation would be fully automatic.

Additionally to the code, the documentation is crosscompiled as well. Furthermore, references are added that refer to the corresponding line of code in the original file.

To make the crosscompilation work, we further had to use the following compiler directives:

- - no-package-directories

  Deletes the file structure of the Java code. This was necessary, as with the file structure the imports were not referenced correctly and had to be changed manually. With the given directive, all files were put into one folder.

- - use-arc

  This was necessary as the iOS project uses the ARC (automatic reference counting) feature of the clang compiler.

- - doc-comments

  With this directive, the documentation is crosscompiled as well. Additionally, references are made that refer to the corresponding line of code in the original file.

- - swift-friendly

  This directive is necessary to generate code that facilitates swift importing.

Continuous Integration

Since the refactoring, the CI differs between compiling frontend and backend. Additionally, the crosscompilation of the backend was added to the pipeline as a last step after the tests. The CI pipeline therefore fails whenever the crosscompilation of the backend is not successful. This ensures that a crosscompilable version is always accessible. After the crosscompilation, the crosscompiled code can be downloaded from the CI as an artifact.

Map representation

In this chapter we will discuss the procedure regarding the display of the map. We will first explain the way the maps are displayed in the android application and then discuss in how far this procedure can be taken over to the iOS project. As this caused huge difficulties within the project, we will also talk about alternatives to the approaches we investigated so far.

Map representation in Android

In the Android application, the maps are displayed using the Mapsforge library[2]. We will now

2 https://github.com/mapsforge/mapsforge

explain the way we work with the map data.

The raw map data comes from the building management and is available in the dxf format which is a common standard for CAD data. We automatically convert this dxf data to osm files. Osm is a map format that is used by the OpenStreetMap project[3]. Such files are structured in an xml-like fashion and can be opened and edited with a software called JOSM[4]. In a last step, those osm files are converted to map files with the software Osmosis[5]. This conversion includes a step of highly efficient reduction. The corresponding map file of an osm file with 40MB of size, is of only 50KB. Afterwards those map files are loaded into the application and displayed with the Mapsforge library. This approach has the advantage, that we do not have to store the osm files within the application (that would be about 500MB), but only the map files (which are of size 10MB). A disadvantage of course results from the need to always apply the conversion step from osm to map data after every change.

Crosscompiling Mapsforge

Since there is no version of the Mapsforge library for iOS, we decided on crosscompiling the whole library as well. Our final goal is to then be able to use it in the iOS application in the very same way we already do in the Android application.

Fortunately, we did not have to start from scratch with crosscompiling the Mapsforge library, since there already existed such a project we could take advantage of[6]. We therefore decided to follow the work done there to, hopefully, end up with a usable Objective-C version of the Mapsforge library.

Sadly the original project ended up to be neither complete nor successful. We only realised this after digging deeper into the issues committed by the author. While retracing the work, we found out, that he got stuck at a particular point, where instead of the map, only a grey area was shown. When clicking on this area, a nullpointer exception caused a crash of the application. We were left with two choices, that where either trying to fix the issues with crosscompiling Mapsforge, or search for a substitute for the Mapsforge library. The latter will be discussed in the following subsection.

The remainder of this subsection will be dedicated to our attempts in fixing the issues the author of the crosscompiled Mapsfore library had and in the understanding we gained of Mapsforge and the general process of displaying maps. This could prove useful giving insight into working with operating system or language depending code when crosscompiling.

In order to convert the Android-specific way the Mapsforge-Android library displays maps into Objective-C code that is usable for iOS, we need to understand the steps applied by the Android library.

When the view attempts to display the map data it calls a frame buffer that generates bitmaps which then make up the tiles of the map. This is the final image of the map Android displays.

3 https://www.openstreetmap.de/
4 https://josm.openstreetmap.de/
5 https://wiki.openstreetmap.org/wiki/Osmosis
6 https://github.com/HallM/MapsForge-ios

Luckily the routine that generates these tiles is not platform dependent thus does not have to be crosscompiled.

Whatsoever the important part for this project is the rendering of the tiles since this is in fact platform dependent. This can be accomplished via the Core Graphics library provided by Apple. Core Graphics is a powerful tool with which a developer can render 2D graphics in Objective-C thus it fits the role of a low level library we can use to display the map data or more precisely the tiles generated by the frame buffer.

The project we used as an example utilised exactly this library but since it unfortunately did not render the map correctly we could not find a fix for the former mentioned error.

Nevertheless another big part of Mapsforge that needs to be written completely new in order to work with iOS is the user input. This is of cause also platform dependent and something our example project did neglect. Zooming or scaling and processing clicks by the user on the map into latitude and longitude values still have to be implemented.

This leaves us with three parts of the crosscompilation process as a whole. The platform independet Java code can be crosscompiled without problems but the user interaction and the map rendering have to be implemented from scratch in huge parts. Since we lacked expertise in working with Core Graphics, and the analysis of the Mapsforge library had the risk of being very time consuming, we first tried to find a library that can function as an alternative to Mapsforge.

Alternative libraries

When looking for alternative libraries, we had some demands that had to be fulfilled. First, the most important demand was of course, that the library should be able to display maps in the osm format, or a format osm can easily be converted to. Secondly, the library must provide some support for indoor maps. As most libraries and most map formats focus on outdoor maps, this requirement came close to demanding the library to be able to work with custom maps. Some libraries only work with the standard map data from Google or Bing, but do not allow to use own map files. Therefore, those libraries could not be considered. Furthermore, the license had to fit our project, meaning the library had to be open source and free to use independent of the size of the project, and we laid additional focus on the library project still being updated by the authors. We considered different libraries but only libosmscout and Tangram ES fulfilled our demands in such a way, that it was worth to investigate them further.

We tried to build the libosmscout library first (because libosmscout allows to display osm files without converting them into another format) but failed with errors while building the project we could not solve easily. In particular the dependencies could not be resolved. Due to this core functionalities of the program were not available which prevented the program from being compiled successfully. Therefore we decided to go on with Tangram ES.

While Tangram ES allows the use of custom maps, these maps must be accessed via an URL. Since for this app the map data is stored locally on the device and not on a server we tried accessing the local files using localhost. Unfortunately we were not able to access the map data that way, meaning we did not find a library which was able to easily replace mapsforge.

Final situation

With the end of the project, we had no ability to display maps in the iOS application - neither with a crosscompiled mapsforge, nor with an alternative library.

As Mapsforge has many benefits we take advantage of in the Android project (that are very small map data, the possibility to display custom maps and being highly efficient), we decided to approach the goal of using Mapsforge in the iOS project. To this end, we will crosscompile the Mapsforge-core which holds the core functionality of the library. We know that this is doable, as we already were successful in doing so. The platform-specific parts then have to be implemented in Objective-C manually.

Quantifiable Results

To quantify the gain of productivity the usage of crosscompiling provided us with, we estimate the time we saved with this approach. To this end, we use COCOMO[7]. COCOMO is used in order to estimate the time that has to be planned for a software project of a given size. The formula for COCOMO reads as

$$t = a * KSLOC^{b}$$

where a and b are constants that consider the complexitiy of the code. t then is the estimated time needed for a project with $KSLOC$ thousand lines of code.

If we assign a moderate complexity, programming the 7000 lines of code of the backend would have taken an estimate of

$$3 * 7^{1,12} = 26.5$$

man months of effort. As all team members spent a total of about 11 man months on the project, we can speak of a great reduction in effort. Note that with more experience, using crosscompilation approaches in the future could lead to an even greater benefit in the reduction of effort, as knowledge could be shared over several projects.

The refactoring had a positive influence towards the project of the Android project. We compared the structure at the 24th of October 2019 (Tag 1.0.7_(28)) with the structure at the 21st of February 2019 (Tag 1.1.1_(15)), right before the project started. One had seen that in the refactored structure there is a decent partition between frontend and backend that can not be found in the old structure. In particular, no package from the backend has any dependencies towards any frontend class or package. The total number of classes and packages rose with the refactoring. This is desired, as in the old project structure it appeared, that one class included both the Android specific frontend as well as the backend part. With the refactoring, such a class was split into two.

7 Barry W. Boehm, Software Engineering Economics, Prentice Hall PTR 1981, 978-0138221225

Conclusion

Our project showed, that crosscompiling code in order to reuse it on a different platform can be a reasonable way to reduce effort in a multi-platform project. While we had difficulties with compiling certain components (the Mapsforge library), we could show, that crosscompiling is doable in general. Backend packages such as the tram for example we could crosscompile and then use in the iOS project without any need of re-programming backend parts or adapting them to the new platform.

We further saw, that the process of crosscompiling had great positive influence on the Android project that came in the form of a refactoring with a partition into frontend and backend. The Android project benefits from the demands of the crosscompilation pipeline. The effort of preparing the crosscompilation therefore accounts for improvements of quality in both projects; hence we believe that this effort was worth to be taken. Though, it has to be noted, that when starting a multiplatform project from scratch, it can be beneficial to use code generation right from the beginning. However, for our purposes code generation was no option, as we already had existing code we wanted to keep.

For the further implementation of UniMaps iOS it is necessary to cope with the challenge of displaying maps. That is, one has to either crosscompile mapsforge successfully, or find an alternative library. The latter could also be solved by writing such a library by oneself. However, as mentioned above, we decided to crosscompile the Mapsforge-core and implement the user interaction and map rendering from scratch.