iOS-Apps in Java

Thomas Künneth, Mathema Software GmbH

Das Open-Source-Projekt Multi-OS Engine ermöglicht die Erstellung nativer iOS-Anwendungen in Java. Dieser Artikel zeigt die Nutzung des Frameworks anhand einer einfach nachvollziehbaren Beispiel-App. Er behandelt neben Installationsvoraussetzungen und Setup die Implementierung der Geschäftslogik sowie der Benutzeroberfläche. Das vollständige Projekt steht in einem GitHub-Repository zur Verfügung [1].

Apps für iOS werden in der Regel in Swift geschrieben. Nur selten kommt noch Objective-C zum Einsatz. Dem Entwickler steht mit Xcode eine mächtige IDE zur Verfügung. Sie unterstützt beim Debuggen, Profilen und Testen, archiviert und paketiert die App und lädt sie in den Store hoch. Auch die Verwaltung von Geräte- und Team-Zertifikaten findet in Xcode statt. Apple-typisch ist die Nutzung des Werkzeugs komfortabel und intuitiv. Dennoch ist für einen effizienten Einsatz einiges an Know-how erforderlich. Dies gilt noch viel mehr für die eigentliche Programmierung. Die bloße Kenntnis der Swift-Syntax ist bei Weitem nicht genug. Wie tickt iOS? Welche Klassen gibt es? Wie wird die Benutzeroberfläche erstellt? In einem Unternehmen, das voll auf iOS-Apps setzt, führt an einem entsprechenden Wissenserwerb kein Weg vorbei. Was aber, wenn eine längerfristige Strategie noch nicht festgelegt wurde? Oder wenn abzusehen ist, dass nur wenige Apps entstehen werden? Kommt eine externe Vergabe nicht in Betracht, sollte vorhandenes Wissen so gut wie möglich genutzt werden. In den letzten 20 Jahren war Java eine der am häufigsten genutzten Technologien für die Erstellung von Unternehmensanwendungen. Es liegt also nahe, iOS-Apps in Java zu schreiben. Damit das funktioniert, muss der Java-Quelltext in native ARM-Maschinensprache übersetzt werden. Außerdem werden eine Art Laufzeitumgebung benötigt, die unter anderem die Speicherverwaltung und die Kommunikation mit iOS übernimmt, und natürlich geeignete Klassenbibliotheken. Sicher keine leichte Aufgabe. Im Laufe der Jahre sind mehrere solcher Frameworks entstanden. Das vielleicht bekannteste ist RoboVM. Im Jahr 2015 wurde das als Open Source gestartete Produkt von der damals noch eigenständigen Firma Xamarin gekauft und nach der Übernahme durch Microsoft im Jahr 2016 eingestellt. Bis heute haben ein paar RoboVM-Forks überlebt.

Überblick

Die ursprünglich von Intel entwickelte Multi-OS Engine *[2]* ist ebenfalls Open Source. Sie steht seit 2016 unter der Apache License Version 2.0 zur Verfügung. Die Firma Migeran ist Project Lead und Core Developer. Sie bietet kommerziellen Support, kundenspezifische Entwicklung sowie Schulungen an. Multi-OS Engine basiert auf der aktuellen Android-Laufzeitumgebung ART (Android Runtime) und verwendet

Abbildung 1: Suche nach dem Multi-OS Engine-Plugin in IntelliJ

müssen Sie noch ein Plug-in herunterladen. Unter Android Studio und IntelliJ erreichen Sie den Plug-in-Manager zum Beispiel auf dem Willkommensbildschirm über „Configure/Plugins". Klicken Sie dann auf „Marketplace" und suchen Sie nach „Multi-OS Engine". Ein Klick auf „Install" *(siehe Abbildung 1)* startet den Installationsvorgang.

In Eclipse öffnen Sie mit „Help/Install New Software" den Installationsassistenten, geben bei „Work With" die Adresse *„https://dl.bintray.com/multi-os-engine/eclipse"* ein und drücken dann die Enter-Taste. Setzen Sie nun ein Häkchen vor „Multi-OS Engine", klicken auf „Next" und folgen dann den weiteren Installationsanweisungen.

Nach der Installation des Plug-ins und dem Neustart der IDE können Sie mit den jeweiligen Assistenten Multi-OS-Engine-Projekte anlegen. Wie das in IntelliJ aussieht, ist in den *Abbildungen 2, 3 und 4* zu sehen. Für einen ersten Test bietet sich die Vorlage „Single View Application" an. Auf der zweiten und dritten Seite des Wizard konfigurieren Sie das Projekt, indem Sie beispielsweise den Namen der App und des Projekts, das Java-Basispaket sowie den Speicherort angeben. Nachdem der Assistent beendet wurde, lädt die IDE fehlende Abhängigkeiten und baut die App. Sie können diese nun starten. In IntelliJ geschieht dies am einfachsten durch Anklicken des grünen Play-Buttons in der Toolbar.

dieselben Java-Bibliotheken wie Android. Die Java-Objective-C-Brücke „Nat/J" erlaubt den Zugriff auf beliebigen Objective-C-Code. Auf diese Weise werden native Benutzeroberflächen aus Java heraus angesprochen. Dazu später mehr. Die Entwicklung erfolgt unter macOS oder Windows. In diesem Fall muss ein macOS-Build-Rechner zur Verfügung stehen. Wie dieser eingerichtet wird, kann in der Multi-OS-Engine-Dokumentation nachgelesen werden *[3]*. Der Einsatz unter Linux ist derzeit nicht möglich. Auf Wunsch kann Multi-OS Engine unter macOS mit brew und repo aus dem Source Code gebaut werden *[4]*. Das ist allerdings nicht nötig. Denn als IDE kommen Android Studio, IntelliJ IDEA oder Eclipse ab Version 4.5 zum Einsatz. Die Installation des Frameworks erfolgt bequem mit den Plug-in-Managern der jeweiligen Entwicklungsumgebung.

Sicher ist Ihnen aufgefallen, dass im vorherigen Absatz öfter das Wort Android gefallen ist. Wieso kommt Googles mobile Plattform ins Spiel, wenn es doch eigentlich um native iOS-Anwendungen geht? Android-Apps wurden von Anfang an in Java geschrieben. Deshalb ist Android Studio (wie auch IntelliJ und Eclipse) optimal auf Java eingestellt. Die Laufzeitumgebung ART wurde für die bestmögliche Ausführung von Java-Code auf mobilen Geräten entwickelt, einschließlich Ahead-of-Time-Compiler und hochoptimierter Speicherverwaltung. Auch die Android-Java-Klassenbibliotheken haben sich in Millionen von Apps bewährt. Multi-OS Engine fußt also auf einem grundsoliden Fundament und kann sich auf iOS-spezifische Anpassungen und Erweiterungen konzentrieren, zum Beispiel auf das Zurverfügungstellen von iOS-API-Bindings. Für die Erstellung der Benutzeroberfläche, das Bauen sowie die Signierung und Paketierung der fertigen App wird Apples Xcode mit den dazugehörenden Kommandozeilentools verwendet. Aus diesem Grund ist bei der Entwicklung unter Windows der bereits angesprochene zusätzliche macOS-Rechner erforderlich.

Installation und Projekt-Setup

Nachdem Sie Xcode und eine der von Multi-OS Engine unterstützten Entwicklungsumgebungen installiert haben,

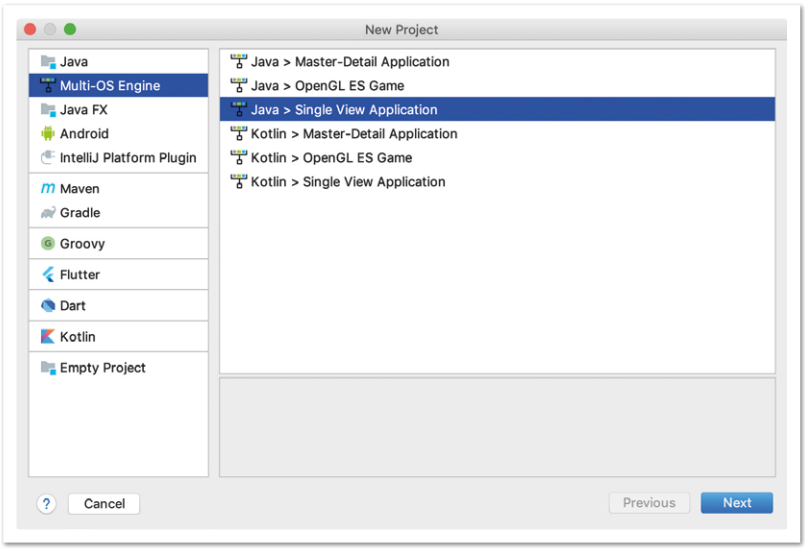

Abbildung 2: Mehrere Projektarten stehen zur Auswahl

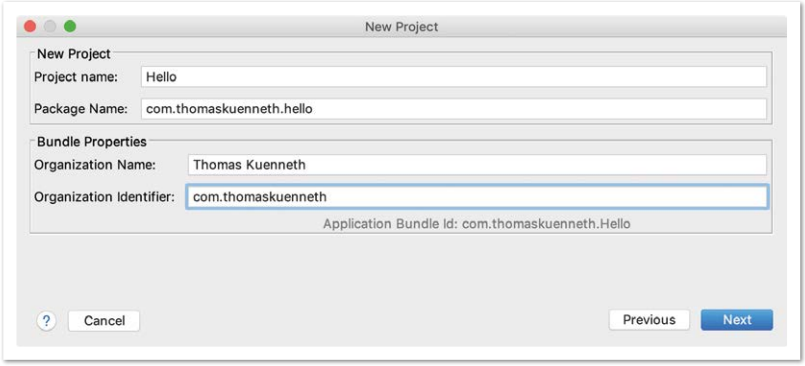

Abbildung 3: App-Informationen erfassen

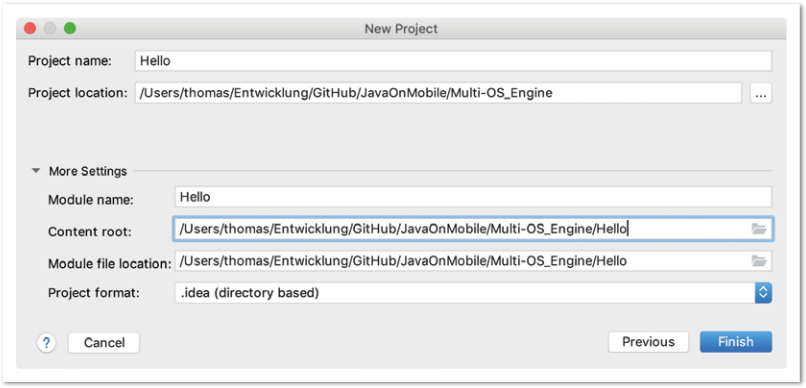

Abbildung 4: Projektname und Speicherort eintragen

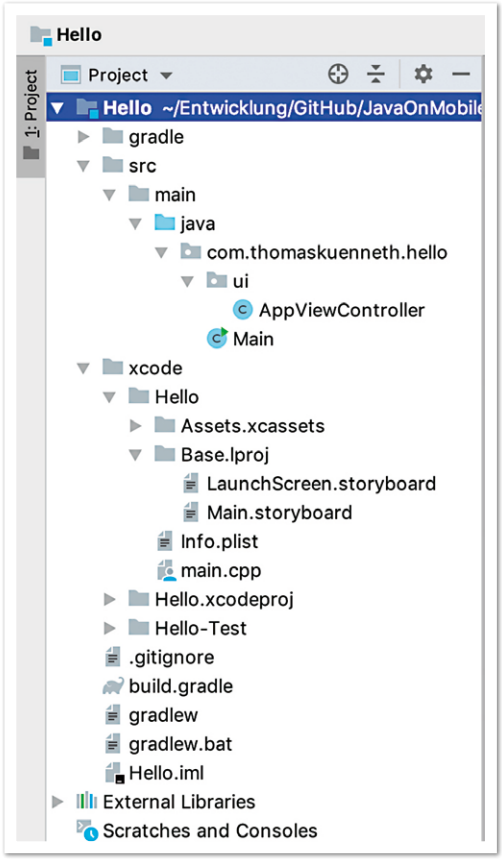

Abbildung 5: Die Projektstruktur

Bitte werfen Sie einen Blick auf die Projektstruktur in *Abbildung 5*. Unterhalb von „src/main/java" befinden sich die beiden Klassen AppViewController und Main. Letztere bildet den Einstiegspunkt in die App und muss im Normalfall nicht verändert werden. AppViewController wird nach dem Hinzufügen von UI-Controls nicht mehr benötigt. Bitte löschen Sie diese dennoch nicht. Sie kann gegebenenfalls als Vorlage für eigene View Controller dienen – dazu gleich mehr. Unterhalb von Xcode befinden sich Dateien, die von Apples IDE verwendet werden. Das ist nötig, weil Multi-OS Engine auf native Werkzeuge zum Bauen der App setzt. Außerdem wird die Benutzeroberfläche in Xcode erstellt. Um das Projekt in Xcode zu öffnen, klicken Sie mit der rechten Maustaste auf „xcode" und wählen dann „Multi-OS Engine Actions/Open Project in Xcode". Falls die Fehlermeldung „Neither the Xcode project nor the workspace is set in the Gradle plugin" erscheint, öffnen Sie das Projekt einfach im Finder. Klicken Sie hierzu mit der rechten Maustaste auf „xcode/Hello" und wählen dann „Reveal in Finder". Ein Doppelklick auf „Hello.xcodeproj" öffnet das Projekt.

Eine Benutzeroberfläche erstellen

In der aktuellen Version enthält die Projektvorlage „Single View Application" keine Bedienelemente. Wir werden im Folgenden eine einfache Benutzeroberfläche erstellen. Bitte legen Sie zunächst mit „File/New/File" eine neue „Cocoa Touch Class" an. Geben Sie dieser den Namen MyViewController. Sie leitet von UIViewController ab und muss in Objective-C implementiert werden. Vor „Also create XIB file" darf kein Häkchen gesetzt sein. Achten Sie darauf, dass die Datei im selben Verzeichnis wie „main.cpp" abgelegt wird und dem Target „Hello" zugeordnet ist. Beides sollte standardmäßig eingestellt sein. Xcode wird nun die beiden Files „MyViewController.h" und „MyViewController.m" erstellen. Als Nächstes müssen Sie den neu erstellten View Controller dem sogenannten „Storyboard" zuordnen. Klicken Sie hierzu als Erstes auf die Datei „Main.storyboard". Im Anschluss daran markieren Sie den Knoten „App View Controller Scene". Wie in *Abbildung 6* zu sehen, müssen Sie im Identity Inspector unter „Custom Class/Class" den neu angelegten MyViewController eintragen. Der Name des Knotens ändert sich zu „My View Controller Scene".

Anschließend müssen Sie aus der Komponentenbibliothek, die mit „View/Libraries/Show Library" geöffnet wird, einen Button sowie ein Label auf das Storyboard ziehen. Danach haben Sie es fast geschafft. Nun definieren Sie für den Button eine sogenannte „Action" und für das Label ein „Outlet". Solche Verbindungen ermöglichen den Zugriff auf die UI-Elemente in Ihrem Code. Klicken Sie auf „View/Assistant Editor/Show Assistant Editor". Neben dem Storyboard sehen Sie nun einen Texteditor. Er enthält in seinem oberen Bereich eine Brotkrümelnavigation. Mit dieser wechseln Sie zur Datei „MyViewController.h". Um ein Outlet oder eine Action zu erstellen, halten Sie die Steuerungstaste gedrückt und ziehen dann das entsprechende UI-Element auf den Assistant Editor *(siehe Abbildung 7)*. Es erscheint ein Pop-up-Menü, mit dem Sie die Verbindung konfigurieren. Legen Sie auf diese Weise eine Action für den Button und ein Outlet für das Label an *(siehe Abbildung 8 und 9)*.

Logik in Java

Bevor Sie Xcode beenden, können Sie dem Button einen aussagekräftigeren Namen geben. Damit die Benutzeroberfläche auf allen möglichen Geräten richtig dargestellt wird, müssen unter Umständen sogenannte „Layout Constraints" angepasst oder hinzugefügt werden. Dies ist ein umfangreicheres Thema und wird im Folgenden nicht weiter dargestellt. Alternativ ist es für dieses Beispiel ausreichend, das Label etwas größer zu ziehen, damit der anzuzeigende Text vollständig dargestellt wird. Speichern Sie Ihre Änderungen und verlassen dann Xcode. Um auf die eben angelegten Verbindungen zuzugreifen, legen Sie die Klasse MyViewController *(Listing 1)* an.

MyViewController referenziert eine Reihe von iOS-spezifischen Klassen, unter anderem UIViewController, UIButton und UILabel. Um gute Benutzeroberflächen erstellen zu können, müssen Sie sich mit deren Verwendung vertraut machen (das gilt auch für die Verwendung von Xcode und des Storyboard-Designers). Zum Beispiel wird die mit `@Override` versehene Methode viewDidLoad()aufgerufen, wenn eine View geladen wurde. Sie eignet sich deshalb prima für bestimmte Initialisierungsaufgaben. Im Vergleich dazu sind die Multi-OS-Engine-spezifischen Ergänzungen schnell erlernbar. Mit der Annotation `@Selector` kennzeichnen Sie Elemente aus nativen Klassen, zum

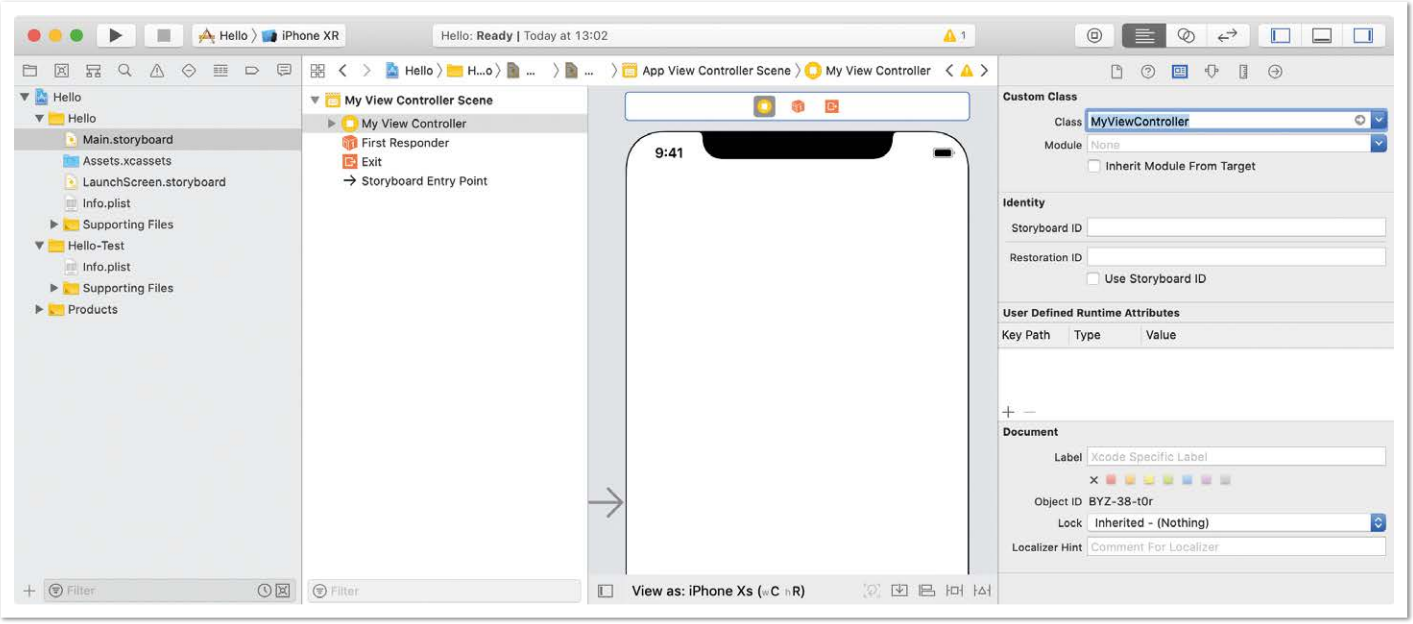

Abbildung 6: Einen neuen View Controller zuordnen

Beispiel die Action der Schaltfläche und das Outlet des Labels. In jedem Fall lohnt ein Blick auf die Dokumentation und das Forum *[5]*. Dort kann unter anderem die Erstellung eigener Bindings nachgelesen werden.

Zusammenfassung

Die Integration echter nativer Benutzeroberflächen ist zweifellos ein großer Pluspunkt. Auf der anderen Seite ist gerade der Wunsch, kein oder nur wenig natives Wissen aufbauen zu müssen, ein Grund für die Suche nach Alternativen. Insofern wirbt Multi-OS Engine auch mit dem Argument, vorhandene Android-Apps als Basis zu nehmen. Die Idee ist, die beiden nativen Teile sowie die gemeinsame Geschäftslogik in separate Module zu packen. Wie das funktioniert, ist in der Dokumentation beschrieben. Eine solche Aufteilung findet sich übrigens in den meisten aktuellen Cross-Platform-Frameworks wieder. Auch der Zugriff auf die jeweiligen nativen Build Tools und die Notwendigkeit mindestens eines Macs sind bewährte Praxis.

Bleibt also die Bewertung der Multi-OS Engine selbst. Die Integration in weitverbreitete IDEs macht die Entwicklung angenehm und komfortabel. Dokumentation und Beispiele sind anschaulich und helfen bei der Einar-

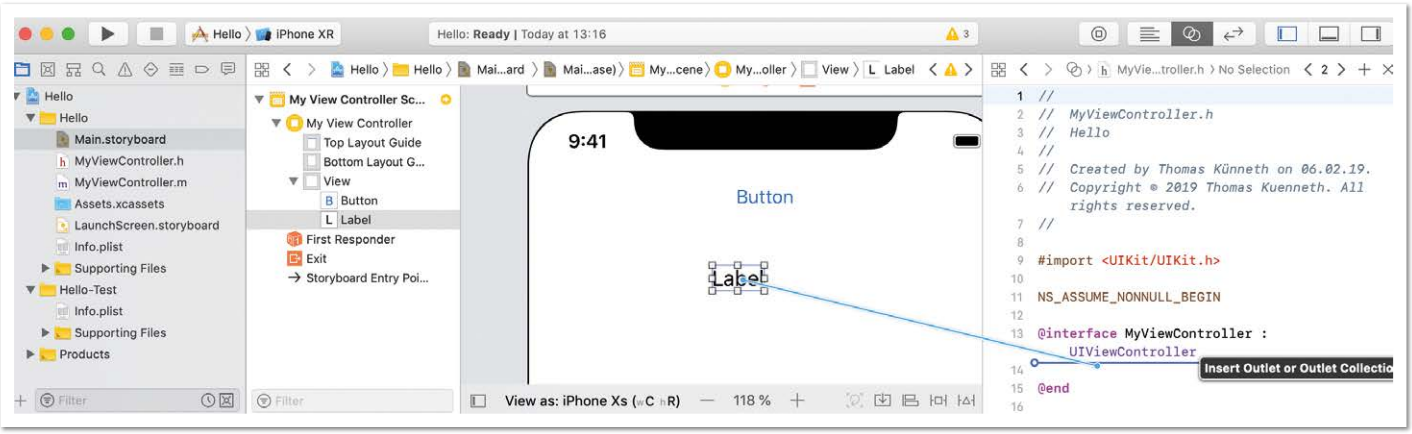

Abbildung 7: Eine Verbindung anlegen

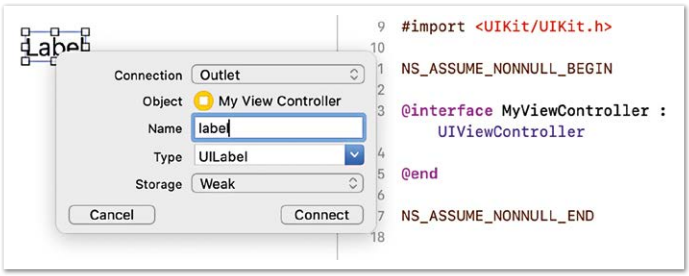

Abbildung 8: Ein Outlet konfigurieren

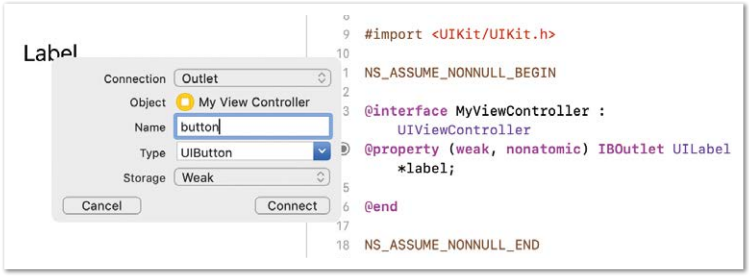

Abbildung 9: Eine Action konfigurieren

```
package com.thomaskuenneth.ui;

import apple.uikit.UIButton;
import apple.uikit.UILabel;
import apple.uikit.UIViewController;
import org.moe.natj.general.Pointer;
import org.moe.natj.general.ann.Owned;
import org.moe.natj.general.ann.RegisterOnStartup;
import org.moe.natj.objc.ObjCRuntime;
import org.moe.natj.objc.ann.IBOutlet;
import org.moe.natj.objc.ann.ObjCClassName;
import org.moe.natj.objc.ann.Property;
import org.moe.natj.objc.ann.Selector;
import java.util.Locale;

@org.moe.natj.general.ann.Runtime(ObjCRuntime.class)
@ObjCClassName("MyViewController")
@RegisterOnStartup
public class MyViewController extends UIViewController {
  public UILabel label = null;
  private int count;

  @Override
  public void viewDidLoad() {
     label = getLabel();
     label.setText("Noch nicht geklickt");
     count = 0;
  }

  @Selector("label")
  @Property
  @IBOutlet
  public native UILabel getLabel();

  @Owned
  @Selector("alloc")
  public static native MyViewController alloc();

  @Selector("init")
  public native MyViewController init();

  protected MyViewController(Pointer peer) {
     super(peer);
  }

  @Selector("button:")
  public void buttonClicked(UIButton sender) {
     label.setText(String.format(Locale.US,
        "%d mal geklickt",
        ++count));
  }
}
```

Listing 1: Die Klasse „MyViewController"

beitung. Sie sind aber leider nicht in allen Punkten aktuell. Gegebenenfalls ist deshalb Forum-Recherche erforderlich. Die kontinuierliche Weiterentwicklung von Android und iOS macht regelmäßige Aktualisierungen von Cross-Platform-Frameworks erforderlich. Denken Sie an neue APIs. Außerdem „schrauben" Apple und Google gerne an den Tool-Schnittstellen. Wenn sich hier inkompatible Änderungen ergeben, bricht der Bauvorgang ab. Probleme kann es auch bei der IDE-Integration geben. Im schlechtesten Fall funktionieren Plug-ins nicht mit aktuellen Versionen einer Entwicklungsumgebung. Hier hilft leider nur ausprobieren.

Insbesondere in der zweiten Jahreshälfte 2018 war es recht still um Multi-OS Engine. Daher bleibt abzuwarten, ob und wann die nächste Version erscheint. Da das Framework quelloffen ist, kann man bei Bedarf einen Blick hinter die Kulissen werfen und zumindest theoretisch selbst Erweiterungen vornehmen oder Fehler beheben.

Links

[1] *https://github.com/tkuenneth/JavaOnMobile/tree/master/Multi-OS_Engine/Hello*

Technologieauswahl

Dieser Artikel geht davon aus, dass eine App ausschließlich unter iOS betrieben wird oder eine bereits vorhandene native Android-Anwendung auf iPhones oder iPads portiert werden soll. Bei der Neuentwicklung einer mobilen App für beide Ökosysteme können „klassische" Cross-Platform-Frameworks eine interessante Alternative sein. Sie versprechen die Entwicklung von in Stores hochladbaren Apps mit einheitlicher Code-Basis. Erreicht wird dies durch eine gemeinsame Programmiersprache, Klassenbibliothek und Werkzeuge sowie eine abstrahierte Sicht auf die Benutzeroberfläche. Wie nativ eine solche App ist, hängt von der Funktionsweise des Frameworks ab. Wird Quellcode transpiliert oder zur Laufzeit interpretiert? Entstehen aus der generalisierten Oberflächenbeschreibung native UI-Elemente oder werden die Widgets selbst gezeichnet? Wie vollständig ist der Zugriff auf native Ressourcen? Wieviel Overhead bringt das Framework mit sich, das heißt, um wie viel größer werden Apps? Eine Technologie, die „weniger native" Apps erzeugt, ist nicht zwangsläufig schlechter geeignet. Denn die Nähe zum Original steht in einem Spannungsverhältnis zu Kosten und vorhandenem Know-how. Hat ein Unternehmen noch keine Software in Swift entwickelt, muss es die Implementierung entweder extern vergeben, Wissen einkaufen oder aufbauen. Ob Letzteres lohnt, hängt unter anderem davon ab, wie viele weitere Apps entstehen sollen. Das gilt natürlich auch für die Programmiersprache eines Cross-Platform-Frameworks. In diesem Bereich dominieren derzeit JavaScript/TypeScript und C#. Ist entsprechendes Wissen nicht vorhanden, muss es eingekauft oder aufgebaut werden. Eine Ausnahme bildet Gluon Mobile. Dieses Framework setzt auf Java und JavaFX.

[2] *https://multi-os-engine.org/*
[3] *https://doc.multi-os-engine.org/multi-os-engine/3_getting_started/3_remote_build/remote_build.html*
[4] *https://github.com/multi-os-engine/multi-os-engine*
[5] *https://discuss.multi-os-engine.org/*

Thomas Künneth
thomas.kuenneth@mathema.de

Thomas Künneth ist Principal Consultant und Head of Mobile bei der MATHEMA Software GmbH. Seine Schwerpunkte sind die Architektur großer Unternehmensanwendungen, Native und Cross-Platform-Apps sowie das Mobile Enterprise. Seit annähernd 20 Jahren beschäftigt sich Thomas intensiv mit Java. Außerdem ist er Android-Entwickler der ersten Stunde und Autor zahlreicher Artikel und Bücher zu Java, Eclipse, Android und Mobile Computing. Thomas spricht regelmäßig auf nationalen und internationalen Konferenzen.