Einstieg in SwiftUI

thomas SILLMANN

User Interfaces erstellen für
macOS, iOS, watchOS und **tvOS**

HANSER

HANSER

Thomas Sillmann

Einstieg in SwiftUI

User Interfaces erstellen für
macOS, iOS, watchOS und tvOS

Der Autor:
Thomas Sillmann, Aschaffenburg
www.thomassillmann.de

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

© 2020 Carl Hanser Verlag München, www.hanser-fachbuch.de
Lektorat: Sylvia Hasselbach
Copy editing: Walter Saumweber, Ratingen
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos

Print-ISBN: 978-3-446-46362-2
E-Book-ISBN: 978-3-446-46587-9

E-Pub-ISBN: 978-3-446-46648-7

Update inside.
Mit unserem kostenlosen Update-Service zum Buch erhalten Sie aktuelle Infos zu den Neuerungen von SwiftUI.

Und so funktioniert es:

1. Registrieren Sie sich unter:
www.hanser-fachbuch.de/swiftui-update

2. Geben Sie diesen Code ein:

4kES-HaN7-aF9P-k87w

Der Update-Service läuft bis Oktober 2022.
Als registrierter Nutzer werden Sie in diesem Zeitraum persönlich per E-Mail informiert, sobald ein neues Buch-Update zum Download verfügbar ist.
Wenn Sie Fragen haben, wenden Sie sich gerne an:
swift-update@hanser.de

Für meine Mutter und meine Schwester.
Danke für all eure Unterstützung und euren kreativen Input.

Inhalt

Titelei

Impressum

Inhalt

Vorwort

1 Über SwiftUI

1.1 Programmierung mit SwiftUI

1.2 Die Preview

1.3 Voraussetzungen

1.4 Integration

2 Grundlagen

2.1 Das View-Protokoll

2.1.1 Ablauf der View-Generierung

2.1.2 Structure vs. Klasse

2.2 Grundlagen der View-Erstellung

2.2.1 Text und Image

2.2.2 Views organisieren mittels Stacks

2.2.3 Views mittels Modifier anpassen

2.2.3.1 Funktionsweise von Modifiern

2.2.3.2 Auszug verfügbarer Modifier

2.2.4 Einsatz von Library und Preview

2.2.5 Layout-System

2.3 Status

2.3.1 Property

2.3.2 State

2.3.3 Binding

3 Views, Controls und Container

3.1 Text und Grafiken

3.1.1 Text

3.1.2 TextField

3.1.3 SecureField

3.1.4 TextEditor

3.1.5 Image

3.2 Buttons

3.2.1 Button

3.2.2 EditButton

3.2.3 PasteButton

3.2.4 Menu

3.2.5 Weitere Buttons

3.3 Value Selectors

3.3.1 Toggle

3.3.2 Picker

3.3.3 DatePicker

3.3.4 Slider

3.3.5 Stepper

3.4 Value Indicators

3.4.1 ProgressView

3.4.2 Label

3.4.3 Link

3.5 Stacks

3.5.1 HStack

3.5.2 VStack

3.5.3 ZStack

3.5.4 LazyHStack und LazyVStack

3.6 Grids

3.7 Listen und Scroll-Views

3.7.1 List

3.7.2 ForEach

3.7.3 ScrollView

3.8 Container-Views

3.8.1 Form

3.8.2 Group

3.8.3 GroupBox

3.8.4 Section

3.9 Weitere Views

3.9.1 Spacer

3.9.2 Divider

4 Navigation und Präsentation

4.1 NavigationView

4.1.1 Grundlagen

4.1.2 Festlegen einer Standardansicht für die Ziel-View

4.1.3 Ändern des NavigationView-Styles

4.1.4 Setzen eines NavigationView-Titels

4.1.5 Navigation-Bar ausblenden

4.1.6 Setzen von Navigation-Bar-Items

4.1.7 Alternatives Auslösen eines NavigationLink

4.1.8 Navigationsstrukturen unter watchOS

4.2 TabView

4.2.1 Grundlagen

4.2.2 Programmatisches Wechseln eines Tab-Bar-Items

4.3 HSplitView und VSplitView

4.4 Sheet

4.4.1 Sheet auf Basis eines Boolean

4.4.2 Sheet auf Basis eines Identifiable-Items

4.4.3 Reaktion auf Ausblenden eines Sheets

4.5 Alert

4.5.1 Erstellen eines Alert

4.5.2 Einblenden eines Alert auf Basis eines Boolean

4.5.3 Einblenden eines Alert auf Basis eines Identifiable-Items

4.6 ActionSheet

4.6.1 Erstellen eines ActionSheet

4.6.2 Einblenden eines ActionSheet auf Basis eines Boolean

4.6.3 Einblenden eines ActionSheet auf Basis eines Identifiable-Items

5 Status

5.1 Property

5.2 State

5.3 Binding

5.4 ObservedObject

5.4.1 Datenmodell vorbereiten

5.4.2 Datenmodell in SwiftUI-View einbinden

5.4.3 Auf Änderungen reagieren

5.5 StateObject

5.6 EnvironmentObject

5.7 Environment

5.8 Zusammenfassung: Welcher Status für welche Situation?

6 Integration

6.1 Hosting

6.1.1 NSHostingController und UIHostingController

6.1.2 WKHostingController

6.2 Representables

6.2.1 Erstellen einer Representable-View

6.2.2 Aktualisieren einer Representable-View

6.2.3 Weitergabe von Aktualisierungen an SwiftUI

7 Preview und Xcode

7.1 Funktionsweise der Preview

7.2 Arbeiten mit der Preview

7.3 Konfiguration der Preview

7.4 Mehrere Previews parallel einsetzen

7.5 Preview ausführen

7.6 Preview auf Device ausführen

7.7 Library

7.8 Kontext-Actions

Nachwort

Vorwort

Liebe Leserin, lieber Leser,

ich entwickle seit inzwischen über zehn Jahren Apps für die verschiedenen Plattformen von Apple. Die Einführung des iPhone und des App Store hat meinen Werdegang maßgeblich beeinflusst und sorgte dafür, dass ich mich heute voll und ganz dem Apple-Kosmos verschrieben habe.

In all diesen Jahren gab es viele kleine Evolutionen, die uns App-Entwicklern das Leben erleichterten. Die Einführung von Automatic Reference Counting vereinfachte die Speicherverwaltung deutlich. Storyboards öffneten ganz neue Wege, App-Strukturen umzusetzen und Views zu gestalten. Auto Layout verbesserte die Möglichkeiten, Views für verschiedene Bildschirmgrößen zu optimieren.

Daneben gab es auch einige wenige *große* Revolutionen. Eine davon war die Einführung der Programmiersprache Swift. Eine andere zeichnet sich erst seit jüngster Zeit ab. Die Rede ist von *SwiftUI*.

Mit SwiftUI ändert sich maßgeblich, wie Views für die verschiedenen Plattformen von Apple umgesetzt werden. Es gibt keine View-Controller mehr, nur Views. Die basieren auf

Structures, nicht auf Klassen. Ihre Erstellung erfolgt deklarativ, nicht imperativ. Und ein Status bestimmt, welches Verhalten sie an den Tag legen und unter welchen Bedingungen sie sich aktualisieren.

Die Arbeit mit SwiftUI ist so gänzlich anders als das, was man all die letzten Jahre mit App-Kit, UIKit und WatchKit gewohnt ist. Gleichzeitig zeichnet sich jetzt bereits ab, wie mächtig dieses neue UI-Framework von Apple ist. Noch nie war es leichter, ansprechende Nutzeroberflächen zu erstellen. Und noch nie brauchte es dafür so wenige Zeilen Code wie mit SwiftUI.

Dazu kommt, dass SwiftUI auf allen Apple-Plattformen zur Verfügung steht. Hat man die grundlegende Funktionsweise demnach einmal verinnerlicht, ist man imstande, Views für macOS, iOS (und iPadOS), watchOS sowie tvOS zu erstellen. SwiftUI stellt ein gemeinsames Toolset dar, das sich im gesamten Apple-Kosmos nutzen lässt.

Seit der erstmaligen Vorstellung von SwiftUI auf der WWDC 2019 bin ich begeistert von diesem Framework. Wie mächtig es ist, wird mir jedes Mal bewusst, wenn ich in Projekten auf die „alten“ Techniken zur Erstellung von Nutzeroberflächen mittels Storyboards und View-Controllern zurückgreife. Im Vergleich ist die Arbeit mit SwiftUI um so vieles komfortabler.

SwiftUI stellt die Zukunft der UI-Erstellung für Apple-Plattformen dar, und mit diesem Buch möchte ich Ihnen einen passenden Einstieg zur Verfügung stellen. In den folgenden Kapiteln erfahren Sie, wie SwiftUI funktioniert und welche Views Ihnen zur Verfügung stehen. Auch gehe ich im Detail auf den Status ein und wie er sich auf die Aktualisierung von Ansichten auswirkt. Ebenso kommt die Integration von SwiftUI in bestehende Projekte auf Basis von Storyboards nicht zu kurz.

Zusätzlich erhalten Sie zusammen mit diesem Buch noch Zugriff auf einen ganz besonderen Service: Dank Update inside kommen Sie in den Genuss von Zusatzkapiteln, die nach und nach veröffentlicht werden. Neben weiteren Themen, die es aus Platzgründen nicht mehr in dieses Buch geschafft haben, werden Sie so auch über kommende SwiftUI-Updates informiert. Der Update-Service läuft bis Oktober 2022. Sie werden persönlich von uns benachrichtigt, wenn neue Updates zum Download zur Verfügung stehen. Registrieren Sie sich dazu einfach unter *www.hanser-fachbuch.de/swiftui-update* mit dem Passwort von der zweiten Seite dieses Buches.

Nun bleibt mir nur noch zu sagen, dass ich Ihnen von Herzen viel Freude mit diesem Buch und der Arbeit mit SwiftUI wünsche. Ergänzende Artikel und Videos rund um die Entwicklung für Apple-Plattformen finden Sie auf meinem Blog unter letscode.thomassillmann.de.

Ihr Thomas Sillmann

Aschaffenburg, August 2020

1 Über SwiftUI

Als Apple Developer stand einem in den letzten Jahren ein klares Set an Frameworks und Funktionen zur Verfügung, um User Interfaces für die verschiedenen Plattformen aus dem Hause Apple umzusetzen. Da gibt es einerseits die Storyboards, mit denen sich – dank entsprechender Xcode-Integration – Nutzeroberflächen in einem separaten Editor komfortabel zusammensetzen lassen. Sogar das Verknüpfen mehrerer Views ist über Storyboards möglich, was in Summe den Umfang des zugehörigen Quellcodes massiv reduzieren kann.

Daneben pflegt Apple schon seit Jahren die bekannten UI-Frameworks AppKit, UIKit und WatchKit. Sie alle enthalten essenzielle UI-Elemente und Funktionen für die verschiedenen Apple-Plattformen. So stellt AppKit die Grundlage für alle bisherigen Mac-Apps dar, während UIKit unter iOS, iPadOS und tvOS zum Einsatz kommt. WatchKit schließlich bringt alles mit, um Anwendungen für die Apple Watch entwickeln zu können.

Mit der WWDC 2019 änderte sich dieses über Jahre bereits erfolgreiche Fundament grundlegend. Denn mit SwiftUI stellte Apple damals ein gänzlich neues UI-Framework vor, das bei den Entwicklern ähnlich unvorhergesehen einschlug wie seinerzeit

die erstmalige Präsentation der Programmiersprache Swift (siehe Bild 1.1).

Doch was ist SwiftUI genau? Und was macht es so besonders?

Wie bereits beschrieben stellt SwiftUI ein UI-Framework dar. Es platziert sich als Alternative zu den bestehenden Frameworks AppKit, UIKit und WatchKit, ohne diese zu ersetzen. Hier kommt aber zugleich die erste große Besonderheit von SwiftUI zum Tragen: Mittels SwiftUI lassen sich Nutzeroberflächen für *alle* Apple-Plattformen erstellen. Egal ob Mac, iPhone, iPad, Apple Watch oder Apple TV: SwiftUI unterstützt sie alle!

Damit sinken für Entwickler die Einstiegshürden enorm. Hatte man beispielsweise bisher ausschließlich Apps für das iPhone auf Basis von UIKit programmiert, musste man sich beim Wechsel auf den Mac zunächst mit AppKit vertraut machen. Es galt dann, den Umgang mit den verschiedenen neuen View- und View-Controller-Klassen zu erlernen und sich mit den Besonderheiten des jeweiligen Frameworks auseinanderzusetzen.

SwiftUI löst dieses Problem, zumindest in Teilen. Mit denselben Views und Funktionen lassen sich mittels SwiftUI Nutzeroberflächen für alle Betriebssysteme von Apple erstellen. Hierbei gilt ein essenzieller Grundsatz:

> *Learn once, apply anywhere.*

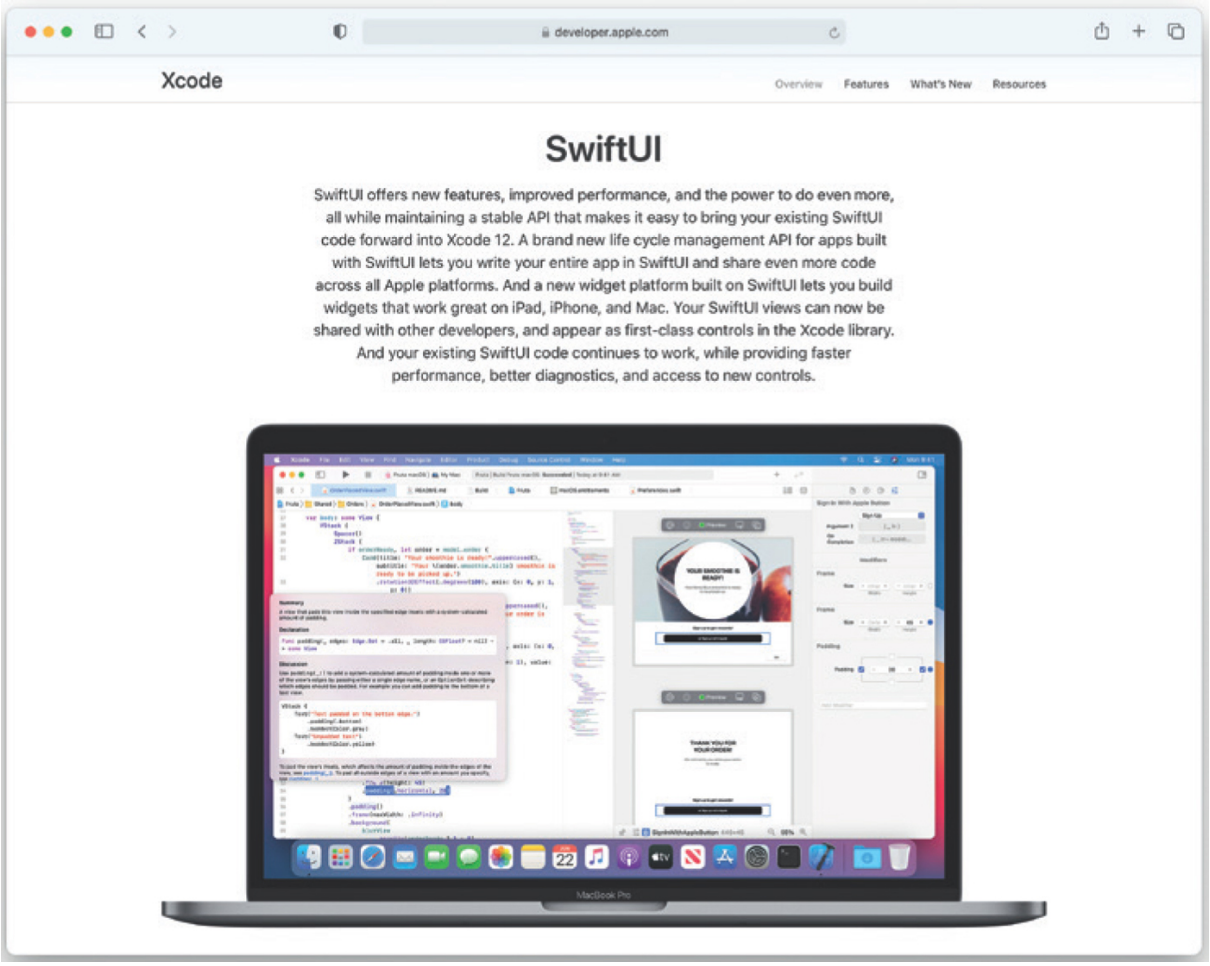

Bild 1.1 SwiftUI war das Highlight der WWDC 2019.

SwiftUI entbindet Entwickler nicht davon, ihre Views für die verschiedenen Plattformen zu optimieren. Man kann sich nur allzu gut vorstellen, dass für die Apple Watch erstellte Nutzeroberflächen eher schlecht als recht für den großen Fernsehschirm geeignet sind.

So geht es bei der Arbeit mit SwiftUI nicht primär darum, einmalig Views zu generieren, die auf allen Apple-Plattformen laufen. Stattdessen stellt SwiftUI ein gemeinsames Toolset dar, das man einmalig lernt, um es dann überall einsetzen zu können.

Aus diesem Grund stehen ein Großteil aller Typen und Funktionen, die innerhalb von SwiftUI definiert sind, sowohl auf dem Mac, dem iPhone, dem iPad als auch auf der Apple Watch sowie auf dem Apple TV zur Verfügung. Das ermöglicht es auch, Views zwischen diesen Plattformen zu teilen (sofern es sinnvoll ist). Beispielsweise ließe sich so eine Zellenansicht innerhalb einer Liste sowohl auf dem Mac, dem iPhone oder der Apple Watch gleichermaßen verwenden, sofern das Aussehen für dieses Element auf allen Plattformen identisch sein soll.

In diesem Zusammenhang ist es aber auch wichtig zu erwähnen, dass nicht alle Elemente innerhalb des SwiftUI-Frameworks auf allen Apple-Plattformen gleichermaßen zur Verfügung stehen. Bestimmte Views und Funktionen lassen sich zum Teil nur unter einzelnen Betriebssystemen nutzen.

Aufschluss hierüber gibt die Xcode-Dokumentation. Zu jedem Element können Sie im oberen rechten Bereich unter der Überschrift *Availability* erkennen, unter welchen Plattformen die jeweilige Funktion zur Verfügung steht (siehe Bild 1.2).

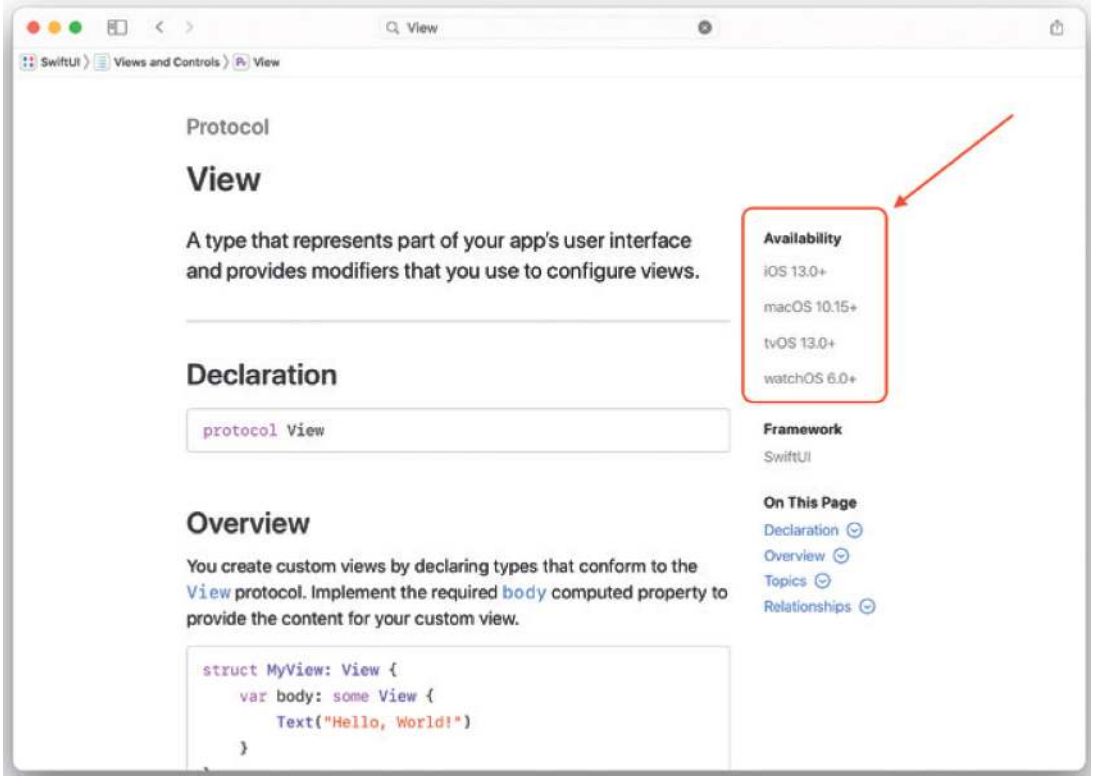

Bild 1.2 Mithilfe der angegebenen SDKs können Sie ermitteln, unter welchen Plattformen die verschiedenen SwiftUI-Funktionen zur Verfügung stehen.

1.1 Programmierung mit SwiftUI

Der Einsatz des SwiftUI-Frameworks unterscheidet sich deutlich von dem, was man bisher von AppKit, UIKit und WatchKit gewohnt ist.

Zunächst wäre da die Syntax. Mit SwiftUI verfolgt Apple einen deklarativen Ansatz zur Erstellung von Views (dem gegenüber steht die imperative Programmierung, wie sie bisher immer zum Einsatz kam).

Statt Views mithilfe von Befehlen à la `addSubview(_:)` zusammenzubauen, legt man das Aussehen und die Struktur in SwiftUI explizit fest. Möchte man beispielsweise ein Label und einen Button untereinander darstellen? Dann packt man beide in einen V-Stack (eine View, um weitere Views vertikal untereinander anzuordnen), und das war's auch schon! Der dafür notwendige Code ist in vereinfachter Form in Listing 1.1 zu sehen.

Listing 1.1 Umsetzung einer einfachen SwiftUI-View mit einem Label und einem Button

```
VStack {
    Text("Hello, SwiftUI!")
    Button(action: {}) {
        Text("Button")
    }
}
```

Diese deklarative Syntax zeigt sehr deutlich, aus welchen Elementen sich eine View zusammensetzt und wie diese angeordnet sind. Beim imperativen Ansatz aus AppKit, UIKit und WatchKit hingegen steuert man mittels Befehlsaufrufen den Aufbau und das Erscheinungsbild von Views. Das kann zu teils fehlerhaften Darstellungen führen, falls diese Aufrufe nicht korrekt oder zu einem falschen Zeitpunkt erfolgen.

SwiftUI ist dank der deklarativen Syntax vor solchen Problemen gefeit, da zu jedem Zeitpunkt klar ist, wie eine View auszusehen hat und wie sie strukturiert ist.

Wie mächtig dieser deklarative Ansatz der View-Erstellung bisweilen sein kann, zeigt der Code in Listing 1.2. Die darin aufgeführten drei Zeilen reichen aus, um eine Table-View-

ähnliche Liste mit SwiftUI zu erstellen, die über 100 Zellen verfügt (siehe Bild 1.3).

Listing 1.2 Erstellen einer Liste mit 100 Zellen.

```
List(0 ..< 100) { row in
    Text("Cell \(row)")
}
```

Hier braucht es keine Implementierung von verschiedenen Data Source-Methoden, wie es beispielsweise bei der Arbeit mit der Klasse `UITableView` unter UIKit der Fall wäre. Darüber hinaus lässt sich dieser kompakte Code sehr gut lesen und er spiegelt genau den Aufbau der Listenansicht wider.

Bild 1.3 Mit drei Zeilen Code lässt sich in SwiftUI eine solche Listenansicht erzeugen.

Verständnis der Listings

Zu diesem Zeitpunkt müssen Sie noch nicht verstehen, was genau innerhalb der bisher gezeigten Listings geschieht. Die Listings sollen vielmehr verdeutlichen, wie die deklarative Syntax von SwiftUI grundlegend aussieht und mit wie wenigen Zeilen Code man bereits beeindruckende Ergebnisse erzielen kann. Selbstverständlich gehe ich in den folgenden Kapiteln detailliert auf die Funktionsweise von SwiftUI ein und erläutere, wie der Code genau funktioniert.

1.2 Die Preview

Eine weitere Besonderheit von SwiftUI und gleichzeitig eines der mächtigsten Features ist die sogenannte *Preview*. Diese erlaubt es, mittels SwiftUI erstellte Views parallel im Editor anzuzeigen (siehe Bild 1.4). So erkennt man auf einen Blick, wie sich Änderungen am Code auf das Aussehen von Ansichten auswirken.

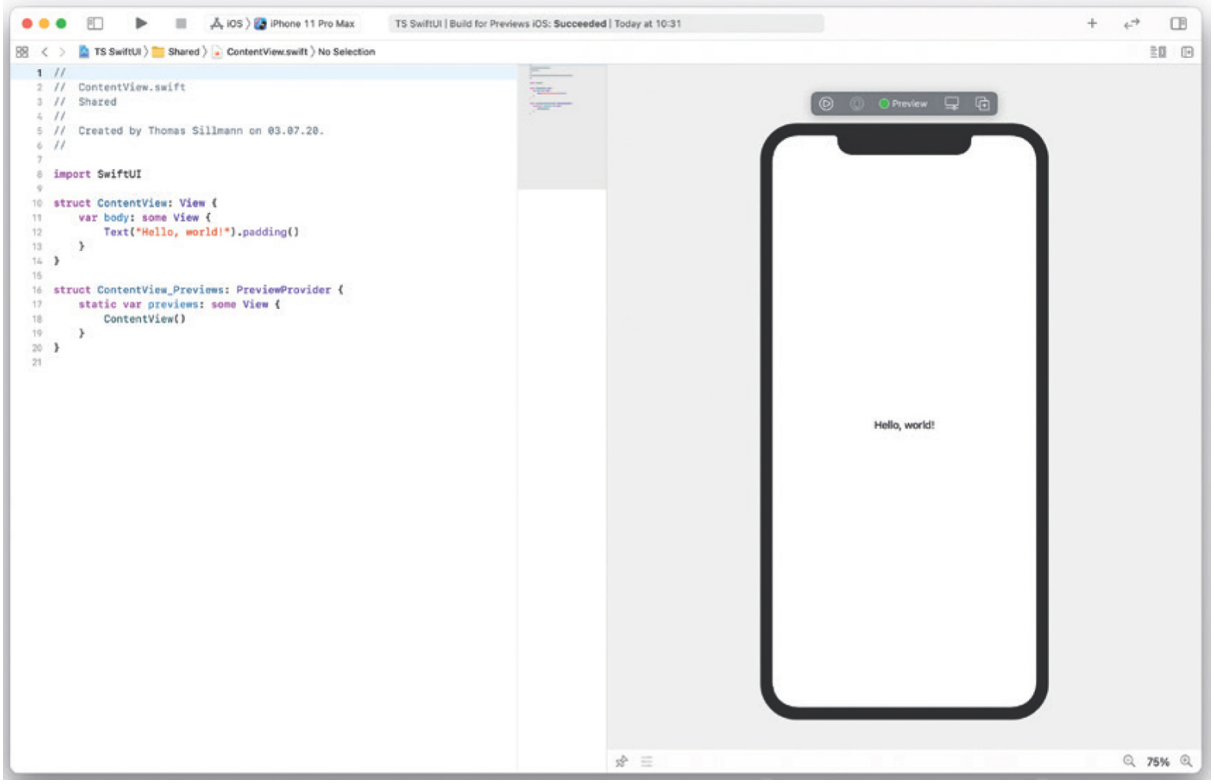

Bild 1.4 Die Preview in der rechten Bildschirmhälfte spiegelt das aktuelle Aussehen unserer SwiftUI-Views wider.

Ehrlicherweise muss man konkretisieren, dass dieses Feature primär der Entwicklungsumgebung Xcode und weniger dem SwiftUI-Framework zu verdanken ist. Apple integrierte diese Vorschaufunktion speziell für SwiftUI in seine IDE, mit dem SwiftUI-Framework selbst hat sie aber im Prinzip nichts zu tun.

Die Preview ist beim Öffnen einer SwiftUI-View standardmäßig immer aktiv. Jedoch ist es notwendig, sie zunächst initial zu starten. Dazu klickt man auf die Schaltfläche mit dem Titel „Resume“ im oberen rechten Bereich der Preview (siehe Bild 1.5).

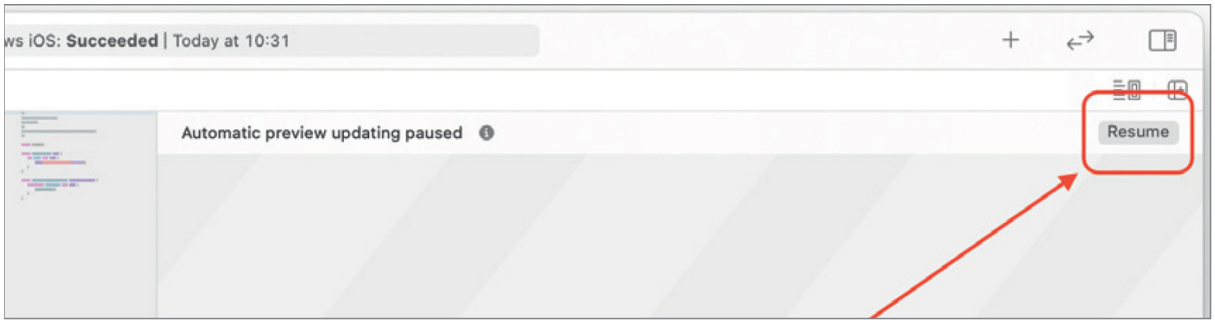

Bild 1.5 Die Preview muss man zunächst per Klick auf die Schaltfläche „Resume“ starten.

Durch Betätigung dieses Buttons findet eine Kompilierung des Xcode-Projekts statt. Das ist notwendig, da Views möglicherweise mit Daten arbeiten, die als eigenständige Typen innerhalb des Projekts definiert sind. Damit die Preview korrekt arbeiten kann, muss sie über den Aufbau dieser Typen und deren Funktionen Bescheid wissen.

Dieser Umstand hat zur Folge, dass auch während der Arbeit an einem Xcode-Projekt die Preview zwischendurch automatisch pausiert. In der Regel geschieht das immer bei Änderungen außerhalb der eigentlichen SwiftUI-View. Auf die kann die Preview ohne Neukompilierung des Projekts nicht zugreifen.

In einem solchen Fall erscheint ebenfalls die in Bild 1.5 zu sehende Leiste am oberen Rand der Preview. Ein Klick auf die Schaltfläche mit dem Titel „Resume“ reicht aus, um erneut die korrekte Vorschau einer SwiftUI-View anzuzeigen.

Ist die Preview einmal aktiviert, aktualisiert sie sich automatisch, sobald man Änderungen an der zugehörigen SwiftUI-View vornimmt. Ändert man so beispielsweise die Größe oder die Farbe eines Textes, ist das geänderte Ergebnis umgehend in der Vorschau ersichtlich.

Doch die Preview kann noch mehr! Sie lässt sich so nicht nur für die Darstellung, sondern auch für die *Änderung* von Views einsetzen. Das funktioniert ganz ähnlich, wie man es beispielsweise von Storyboard-Dateien her kennt. So lassen sich Elemente wie Labels und Buttons innerhalb der Preview auswählen, um dann Anpassungen über den Attributes Inspector vorzunehmen (siehe Bild 1.6). Welche Änderungsmöglichkeiten hierbei konkret zur Verfügung stehen, hängt immer vom ausgewählten Element ab.

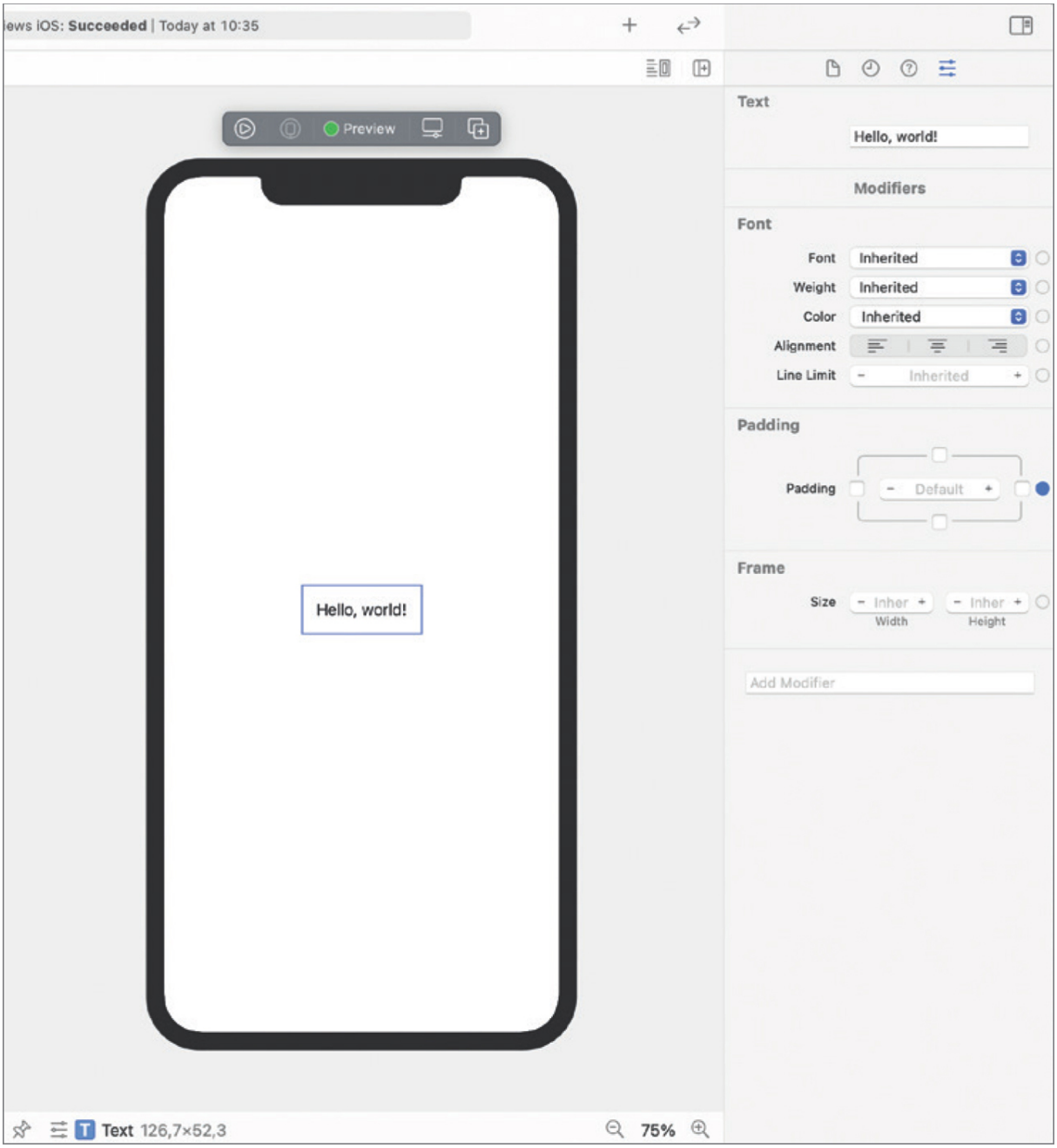

Bild 1.6 Mithilfe der Preview und des Attributes Inspectors lassen sich ebenfalls SwiftUI-Views anpassen.

Darüber hinaus ist es möglich, der Preview neue View-Elemente mittels Drag-and-drop hinzuzufügen. Hierzu öffnet man zunächst die Library über die Plus-Schaltfläche am oberen rechten Rand