Einstieg in SwiftUI

thomas SILLMANN

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

HANSER

Sillmann

Einstieg in SwiftUI

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:

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

Thomas Sillmann

Einstieg in SwiftUI

User Interfaces erstellen für macOS, iOS, watchOS und tvOS

HANSER

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.

Lektorat: Sylvia Hasselbach
Copy editing: Walter Saumweber, Ratingen
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Max Kostopoulos
Satz: Kösel Media GmbH, Krugzell
Druck und Bindung: CPI books GmbH, Leck
Printed in Germany

Print-ISBN: 978-3-446-46362-2
E-Book-ISBN: 978-3-446-46587-9
E-Pub-ISBN: 978-3-446-46648-7

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

Inhalt

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 AppKit, 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.

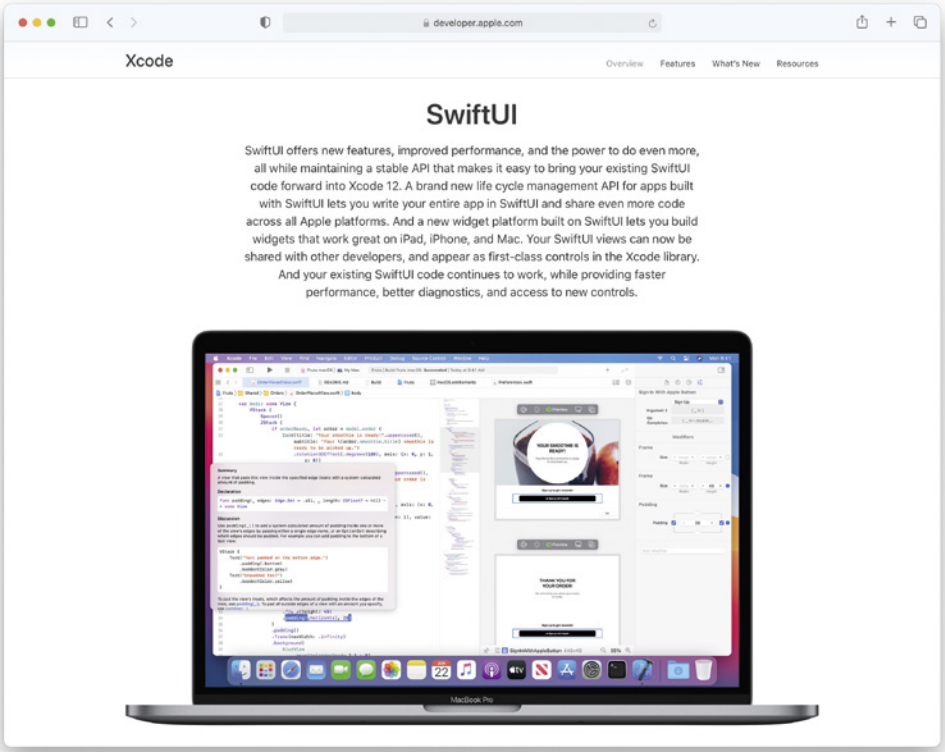

Bild 1.1 SwiftUI war das Highlight der WWDC 2019.

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).

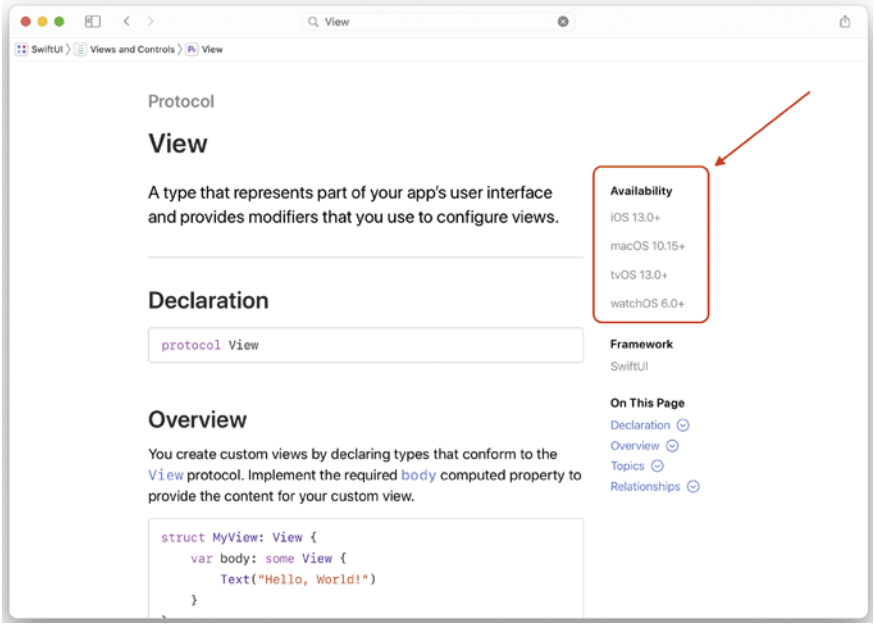

Bild 1.2 Mithilfe der angegebenen SDKs können Sie ermitteln, unter welchen Plattformen die verschiedenen SwiftUI-Funktionen zur Verfügung stehen.

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.

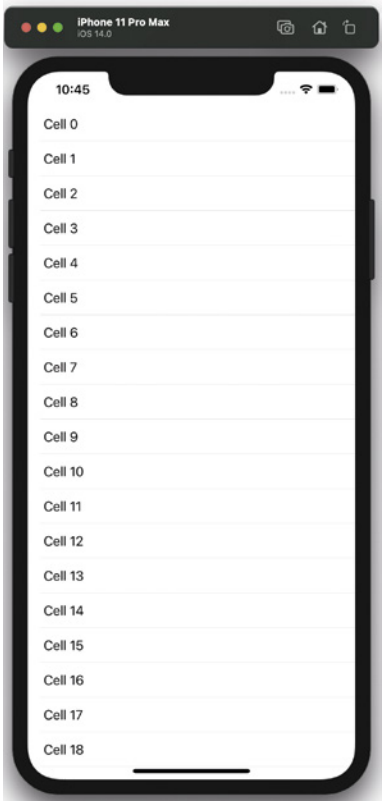

Bild 1.3
Mit drei Zeilen Code lässt sich in SwiftUI eine solche Listenansicht erzeugen.

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.

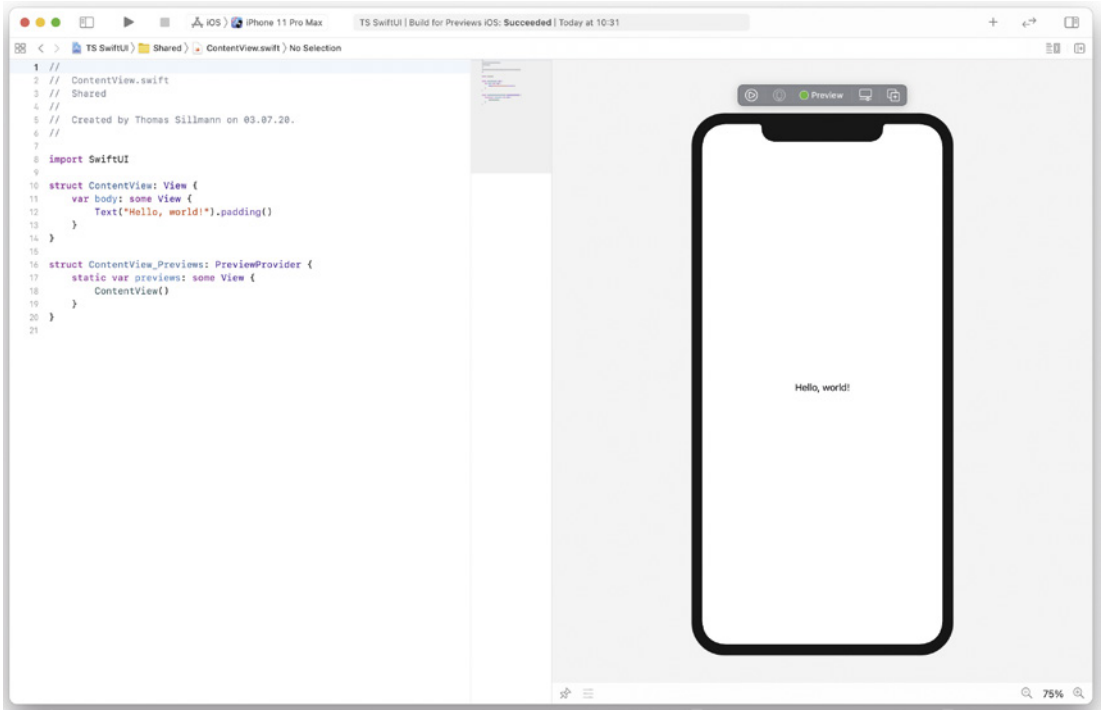

Bild 1.4 Die Preview in der rechten Bildschirmhälfte spiegelt das aktuelle Aussehen unserer SwiftUI-Views wider.

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).

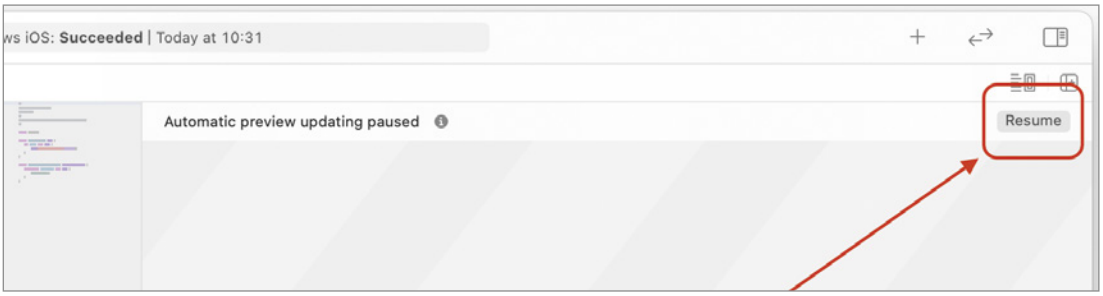

Bild 1.5 Die Preview muss man zunächst per Klick auf die Schaltfläche „Resume“ starten.

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.

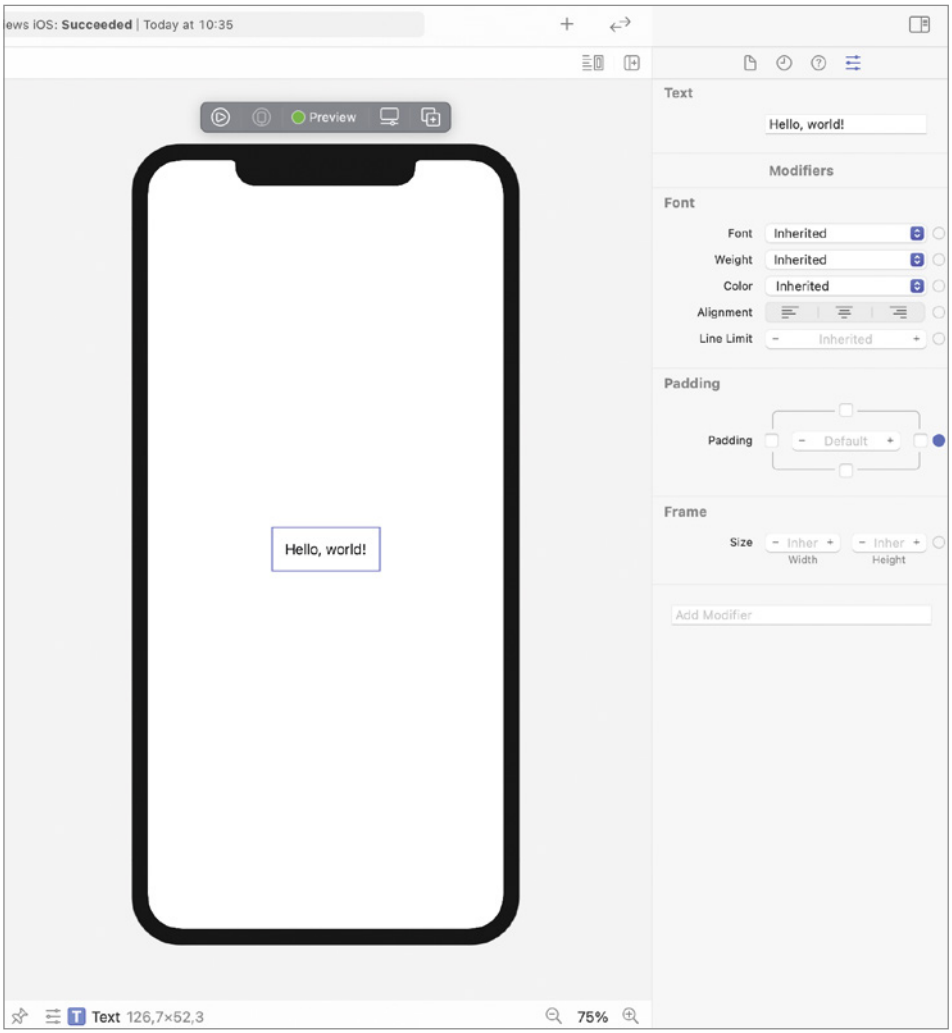

Bild 1.6 Mithilfe der Preview und des Attributes Inspectors lassen sich ebenfalls SwiftUI-Views anpassen.

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 der Xcode-Toolbar. Unter den verfügbaren Reitern wählt man im Anschluss denjenigen links außen aus (die sogenannte *Views Library*, siehe Bild 1.7). Dort findet man eine Auflistung diverser Views, die sich mit SwiftUI einsetzen lassen. Diese können nun aus der Library sowohl in den Code als auch in die Preview gezogen werden, um sie an der entsprechenden Stelle einzufügen.

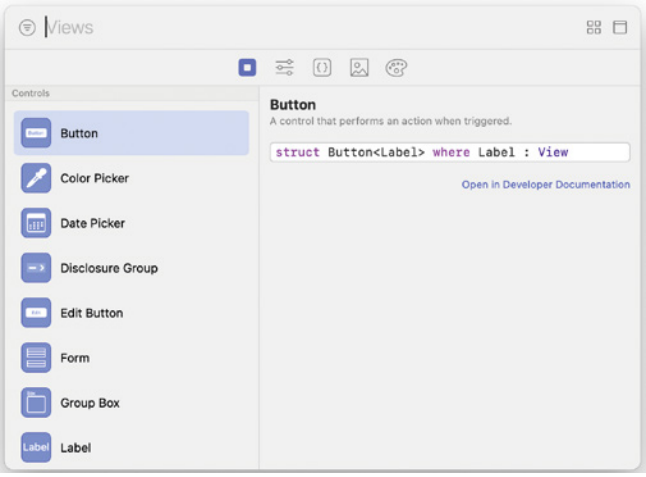

Bild 1.7 Über die Library hat man Zugriff auf verschiedene Elemente des SwiftUI-Frameworks.

Dieser erste grobe Überblick zur Preview soll an dieser Stelle genügen. Mehr zur Preview und die effiziente Arbeit damit erfahren Sie sowohl im nachfolgenden Kapitel als auch in Kapitel 7 („Preview und Xcode“).

1.3 Voraussetzungen

Das SwiftUI-Framework steht nur für die neuesten Versionen der verschiedenen Apple-Betriebssysteme zur Verfügung. Das ist wichtig zu wissen, weil der Einsatz von SwiftUI in bestimmten Projekten so (noch) nicht möglich oder aufwendiger ist.

Im Folgenden erhalten Sie einen kleinen Überblick, welche Voraussetzungen erfüllt sein müssen, damit Sie mit dem SwiftUI-Framework in Ihren Projekten arbeiten können.

Zunächst ist da die Entwicklungsumgebung Xcode. Neben der Preview-Funktion (siehe Abschnitt 1.2, „Die Preview“) bringt sie auch das SwiftUI-Framework mit. Dazu benötigen Sie mindestens Version 11 von Apples IDE. Die aktuellste Version können Sie direkt aus dem Mac App Store herunterladen und installieren (siehe Bild 1.8).

Für die verschiedenen Apple-Plattformen können Sie SwiftUI ab den folgenden Versionen einsetzen:

- macOS: ab Version Catalina 10.15
- iOS: ab Version 13
- iPadOS: ab Version 13
- watchOS: ab Version 6
- tvOS: ab Version 13

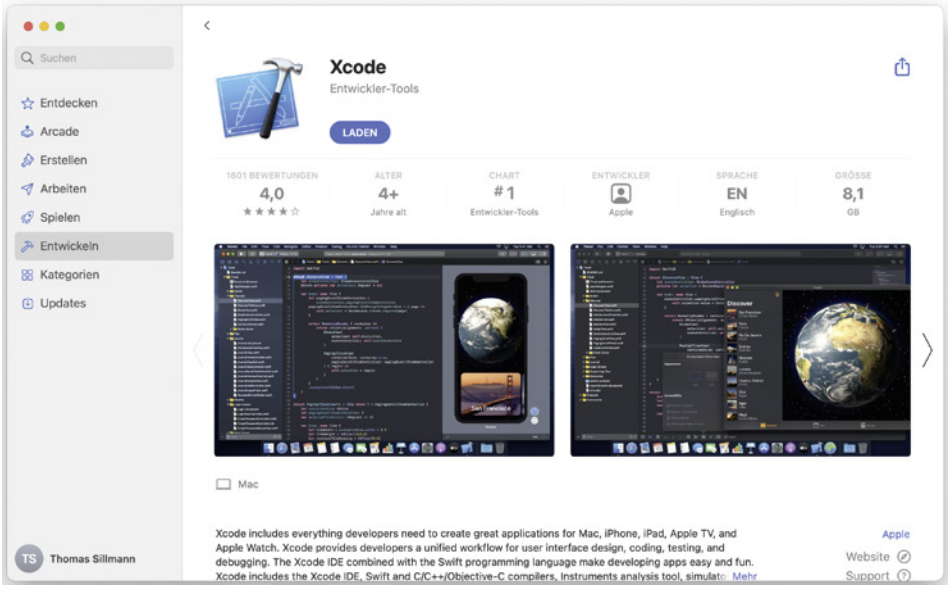

Bild 1.8 Ab Version 11 von Xcode ist SwiftUI in Apples IDE integriert.

Wichtig hierbei: Manche Elemente und Funktionen des Frameworks stehen erst mit neueren Versionen zur Verfügung. Die entsprechende Info können Sie der Dokumentation entnehmen.

Und auch wenn es offensichtlich anmuten mag, möchte ich abschließend noch ein Wort zur Programmiersprache verlieren, die bei der Arbeit mit SwiftUI zum Einsatz kommt. Denn wie kaum anders zu erwarten, lässt sich das SwiftUI-Framework ausschließlich mit Swift nutzen, nicht mit der ebenfalls im Apple-Umfeld bekannten Programmiersprache Objective-C.

1.4 Integration

Auch wenn die im vorherigen Abschnitt genannten Voraussetzungen erfüllt sind, möchte man bereits bestehende Projekte auf Basis von AppKit, UIKit und WatchKit in der Regel nicht komplett auf SwiftUI umstellen. In solchen Szenarien wäre es meist besser, bestehende Views und View-Controller sowie Storyboard-Dateien beizubehalten. Gleichzeitig möchte man womöglich aber auch langfristig die Vorteile von SwiftUI nutzen und so neue View-Elemente vorzugsweise mit SwiftUI erstellen.

Erfreulicherweise ist ein Mischbetrieb von AppKit, UIKit und WatchKit zusammen mit SwiftUI kein Problem. Apple stellt entsprechende Klassen, Protokolle und Funktionen zur Verfügung, um sowohl bestehende Views und View-Controller in SwiftUI einzubinden als auch SwiftUI-Views aus AppKit, UIKit und WatchKit heraus zu nutzen.

Die Entscheidung, SwiftUI einzusetzen, gilt dementsprechend nicht zwingend für ein ganzes Projekt. Man hat zu jeder Zeit die Möglichkeit, Views und View-Controller auch über die „klassischen“ Wege zu erstellen und zu nutzen.

Letztlich ist SwiftUI schlicht ein alternatives UI-Framework, das sich parallel zu AppKit, UIKit und WatchKit positioniert. Ob man damit nun ein ganzes Projekt oder nur einige wenige Teil-Views umsetzt, ist jedem Entwickler vollkommen selbst überlassen.

Alle weiteren Informationen zum Thema Integration finden Sie in Kapitel 6 dieses Buches.

2 Grundlagen

Mit diesem Kapitel geht es nun richtig los! ☺ Nach dem ersten groben Überblick zu SwiftUI im vorangegangenen Kapitel erfahren Sie nun, wie das Framework konkret funktioniert und mithilfe welcher Bestandteile und Techniken Sie Views umsetzen. Hierbei lernen Sie bereits alle grundlegenden Bestandteile und Mechanismen kennen, die SwiftUI so einzigartig machen.

2.1 Das View-Protokoll

Das `View`-Protokoll stellt die Basis einer jeden View in SwiftUI dar. Es definiert die elementaren Eigenschaften und Funktionen, die jede Ansicht mitbringen muss.

In gewisser Weise ist es vergleichbar mit der `UIView`-Klasse aus UIKit beziehungsweise der `NSView`-Klasse aus dem AppKit-Framework. Diese dienen ihrerseits als Basis für alle Ansichten, die über das jeweilige Framework erzeugt werden. Und genauso muss jede View, die wir mittels SwiftUI erstellen, konform zum `View`-Protokoll sein (siehe Bild 2.1).

Die wichtigste Eigenschaft des View-Protokolls ist eine Property namens `body`. Über sie definiert man das Aussehen und die Funktionsweise einer Ansicht. Als Ergebnis liefert `body` ebenfalls ein Element zurück, das konform zum `View`-Protokoll ist.

Lassen Sie uns anhand dieser ersten Informationen direkt ein erstes konkretes Beispiel betrachten. Dazu finden Sie in Listing 2.1 die Umsetzung einer einfachen SwiftUI-View, die ein Label mit dem Text „Hello, World!“ ausgibt. Die View besitzt den Namen `ContentView` und ist als Structure umgesetzt (siehe hierzu auch Abschnitt 2.1.2, „Structure vs. Klasse“). Sie liefert eine Instanz vom Typ `Text` zurück. `Text` ist eine Structure aus dem SwiftUI-Framework und seinerseits konform zum `View`-Protokoll, entsprechend können wir es als Rückgabewert für die `body`-Property nutzen (mehr zu den verfügbaren Views und Controls in SwiftUI erfahren Sie in Kapitel 3 dieses Buches). `Text` ist vergleichbar mit der Klasse `UILabel` aus dem UIKit-Framework und dient zur Darstellung kurzer Texte.

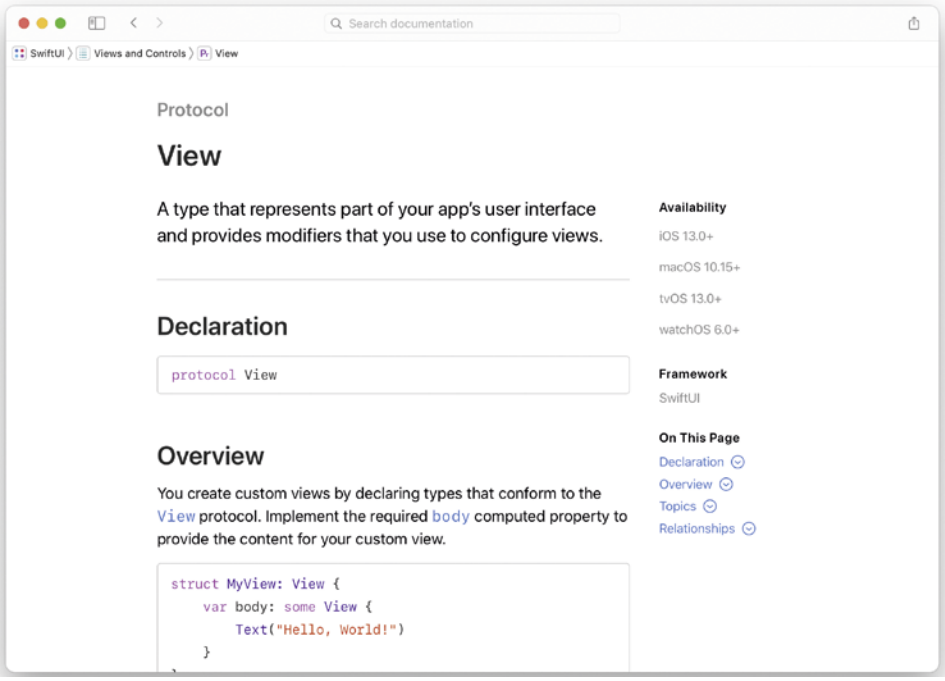

Bild 2.1 Alle Views in SwiftUI basieren auf dem View-Protokoll.

Listing 2.1 SwiftUI-View zur Darstellung eines Labels

```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

Warum kein return?

Womöglich wundern Sie sich, warum das Ergebnis der body-Property – die Text-Instanz mit dem Inhalt „Hello, World!“ – nicht explizit mittels `return` zurückgegeben wird. Hierbei handelt es sich um ein Feature von Swift 5.1, der sogenannten *Shorthand Getter Declaration*. Sie erlaubt es uns – sollte der Getter einer Computed Property aus nur einem einzigen Befehl bestehen, der gleichzeitig das gewünschte Ergebnis generiert – auf die explizite Angabe von `return` zu verzichten.

Der Einsatz der Shorthand Getter Declaration ist in SwiftUI gang und gäbe. Sie trägt ihren Teil dazu bei, den Quellcode von SwiftUI-Views möglichst kompakt und übersichtlich zu halten.

Eine wichtige Rolle bei der Deklaration von SwiftUI-Views spielt der Typ der `body`-Property. Sofern es keinen triftigen Grund gibt (und mir selbst ist ein solcher bisher bei der Arbeit mit SwiftUI noch nicht unterkommen), sollte man diesen immer mittels `some View` deklarieren (so wie auch in Listing 2.1 zu sehen).

Um eines vorneweg klarzustellen: Das Beispiel aus Listing 2.1 funktioniert auch dann tadellos, wenn man als Typ für die `body`-Property explizit `Text` angibt. Es reicht umgekehrt jedoch nicht aus, schlicht das Protokoll `View` für die Typ-Deklaration zu nutzen (siehe hierzu auch Listing 2.2).

Listing 2.2 Deklaration der body-Property

```
// Möglich!
var body: Text {
    // ...
}

// Fehler!
var body: View {
    // ...
}
```

Der Grund ist, dass das `View`-Protokoll einen Associated Type definiert. Hierbei handelt es sich um den Typ jener Ansicht, die man anzeigen möchte (im vorangegangenen Beispiel war das ein Text). `View` benötigt zwingend die Information darüber, welcher konkrete Typ als Associated Type verwendet werden soll, um funktionieren zu können. Definiert man – wie im zweiten Beispiel von Listing 2.2 zu sehen – als Rückgabewert für eine Eigenschaft oder Funktion schlicht den Protokoll-Typ `View`, fehlt jener konkrete Associated Type, und es kommt zu einem Compiler-Fehler.

Doch auch die explizite Angabe des konkreten Typs, den man mittels `body`-Property zurückliefert, ist nicht ideal. Hierbei spielen vorrangig zwei Gründe eine Rolle:

- Gerade bei der Arbeit mit SwiftUI kann es schnell passieren, dass sich der Rückgabetyp einer View ändert. Ergänzt man beispielsweise die Ansicht aus Listing 2.1 noch um ein zusätzliches Bild, das neben dem Label angezeigt wird, gibt die View keine `Text`-Instanz mehr zurück. Stattdessen stellt die View nun einen Stack dar, der einen Text und ein Bild enthält (dazu später mehr). Geben wir nun den Rückgabetyp einer View in der `body`-Property explizit an, müssen wir ihn jedes Mal ändern, wenn wir die Struktur und den Aufbau der Ansicht aktualisieren. Das ist ein Aufwand, den wir uns in der Regel sparen wollen.
- Möglicherweise soll nach außen hin gar nicht ersichtlich sein, welchem konkreten Typ die `body`-Property einer View entspricht. Das ist insbesondere dann ein sehr wichtiger Punkt, falls man ein Framework erstellt und Teile der Implementierung nicht zugänglich sein sollen. In diesem Fall kommt eine explizite Angabe des Rückgabetyps von `body` schlicht nicht in Frage.

Die Lösung für diese Problematik sind die mit Swift 5.1 eingeführten sogenannten *Opaque Types*. Mithilfe eines solchen Opaque Type lässt sich der konkrete Typ, den eine Funktion zurückliefert, verschleiern. Gleichzeitig legt man fest, dass jener konkrete Typ immer identisch ist.

Um einen solchen Opaque Type zu definieren, nutzt man das Schlüsselwort `some`. Genau einen solchen Opaque Type nutzten wir in Listing 2.1 als Rückgabetyp für die `body`-Property:

```
var body: some View { ... }
```

So sind die beiden aufgeführten Probleme gelöst: In `body` können wir nun eine beliebige View zurückgeben, solange diese nur konform zum `View`-Protokoll ist. Da spielt es keine Rolle, ob es sich bei dem Ergebnis nun um einen Text, ein Bild oder eine Liste handelt. Gleichzeitig ist aus der `body`-Deklaration heraus nicht ersichtlich, welche *konkrete* View zurückgeliefert wird. Das ist ausschließlich in der Implementierung von `body` definiert.

Mit SwiftUI erstellte Ansichten deklariert man in der Regel immer mittels des Opaque Type `some View`. Das bietet die größtmögliche Flexibilität und ist die ideale Grundlage für SwiftUI-Views.

2.1.1 Ablauf der View-Generierung

Wird eine SwiftUI-View geladen, führt das automatisch zum Aufruf der zugehörigen `body`-Property. Bei näherer Betrachtung erscheint dieses Konzept aber fehlerträchtig. Denn wie wir wissen, generiert man innerhalb von `body` wiederum selbst eine View. Diese springt entsprechend dann in die Implementierung *ihrer* `body`-Property und so weiter. Es entsteht demnach eine Endlosschleife, die in vereinfachter Form in Bild 2.2 skizziert ist.

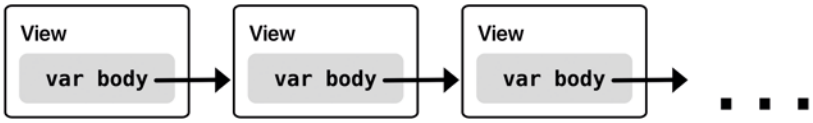

Bild 2.2 Eine View generiert aus ihrer body-Property heraus eine weitere View.

Um dieses Problem zu lösen, stehen innerhalb des SwiftUI-Frameworks sogenannte *Primitive Views* zur Verfügung. Eine Primitive View beendet den beschriebenen Kreislauf und dient als Endpunkt. Sie stellt das letzte Glied in dieser View-Hierarchie dar.

Zu den Primitive Views in SwiftUI zählen unter anderem die Elemente `Text`, `Color`, `Spacer`, `Image`, `Shape` und `Divider`. Mehr zu den verschiedenen Views, die im SwiftUI-Framework enthalten sind, erfahren Sie in den Kapiteln 3 und 4 dieses Buches.

2.1.2 Structure vs. Klasse

Da es sich bei `View` um ein Protokoll handelt, ist es theoretisch möglich, es nicht nur auf Structures, sondern auch auf Klassen anzuwenden. Bei der Arbeit mit SwiftUI kommt das aber nicht infrage. Views in SwiftUI basieren **immer** auf Structures, nicht auf Klassen.

Doch warum ist das so? Aus AppKit, UIKit und WatchKit ist man es schließlich gewohnt, Views und View-Controller immer als Klassen umzusetzen. Das hängt nicht zuletzt mit der Vererbung zusammen. Eine Klasse wie `UIView` bringt so ein grundlegendes Set an Eigenschaften und Funktionen mit, auf dem alle Subklassen aufbauen können.