Deklarative Nutzeroberflächen übernehmen die App-Entwicklung

Wirksam angeleitet

Thomas Künneth

Deklarativen UI-Frameworks gehört die App-Zukunft. Aus gutem Grund: Der Code ist besser zu lesen und zu warten als mit imperativen Ansätzen.

Mit imperativen UI-Frameworks müssen Entwickler alle Schritte, die für das Anzeigen und Verändern der Benutzeroberfläche notwendig sind, manuell programmieren. Ändert sich der Zustand der App, müssen sie den Komponentenbaum anpassen – aufwendig und fehleranfällig. Bei deklarativen Ansätzen beschreibt man hingegen auf Basis des aktuellen Zustands, wie die Oberfläche aussehen soll. Wie das Framework den zuletzt aktuellen Stand entsprechend modifiziert, ist ihm überlassen – im Idealfall erzeugt es einen besser les- und wartbaren Quelltext.

Komponentenbäume strukturieren die Oberfläche

Meist fassen UI-Frameworks Bedienelemente zur Laufzeit zu baumartigen Strukturen zusammen. Je nach Programmiersprache sind deren Knoten einfache Datenstrukturen oder echte Objekte. Die Wurzel repräsentiert eine Seite, ein Fenster oder den Bildschirm. Kinder sind entweder einfache Elemente wie Button, Label, Textfeld oder Grafik – oder sogenannte Container. Diese können – wie die Wurzel – ebenfalls normale Bedienelemente und weitere Container enthalten. Jeder Knoten speichert elementare Werte wie Position, Breite und Höhe. Wie diese berechnet werden, hängt vom verwendeten Framework ab. Häufig übernehmen die Container diese Aufgabe: Sie kümmern sich um das Layout. Weitere Werte der Knoten sind oft elementspezifisch, zum Beispiel der Text eines Buttons oder Labels, seine Farbe oder Schriftart. Um eine Benutzeroberfläche zum Leben zu erwecken, müssen Entwickler das Objektgeflecht zunächst erstellen und anschließend situationsgerecht manipulieren. Daher lassen sich die meisten Attribute der UI-Komponenten lesen und schreiben.

Was das bedeutet, lässt sich anhand eines einfachen Zählers erläutern: Jeder Button-Klick erhöht den Wert einer Variablen und gibt ihn aus. Initial sehen Nutzer wie in Abbildung 1 ein „Noch nicht geklickt“. Listing 1 zeigt eine Umsetzung mit Java Swing, die einen Großteil der Oberfläche in der Methode `createUI()` zusammensetzt und sie in `updateUI()` aktualisiert.

Das Inkrementieren der Zählvariable versteckt sich ebenfalls in `createUI()`. Das Mischen von Oberflächendefinitionen, Verhalten und Zustand macht Programme schnell unübersichtlich und damit fehleranfällig. Deshalb etablierten sich Entwurfsmuster, unter anderem Model-View-Controller (MVC) und Model-View-

ViewModel (MVVM). Sie strukturieren die Anwendung und machen sie besser wartbar.

Oberflächenbeschreibungen auslagern

Statt die Oberfläche durch Ketten von Objekt-Instanziierungen und Setzen ihrer Eigenschaften im Quelltext zu programmieren, kann man sie sich mit entsprechenden Frameworks in eigenen Dateien beschreiben. Zur Laufzeit entfaltet das System diese Dateien dann zu Objektgeflechten. Android, JavaFX, .NET und iOS nutzen hierfür XML. Praktisch alle Plattformen haben mittlerweile komfortable Editoren für die Gestaltung an Bord. Listing 2 zeigt die Oberflächenbeschreibung des Zähler-Beispiels in der Auszeichnungssprache Extensible Application Markup Language (XAML) – das vollständige Universal-Windows-Platform-Beispiel (UWP) und alle anderen Programme stehen auf dem *iX*-Listing-Server bereit (siehe ix.de/z1e9).

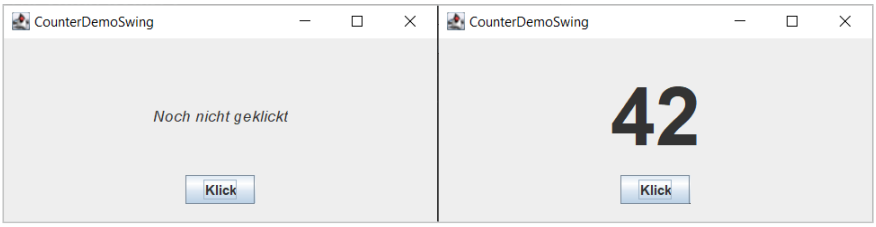
XAML kommt unter anderem bei UWP- und Xamarin-Apps zum Einsatz; die Sprache beschreibt die Oberfläche als baumartige Struktur, wobei die verwendeten XML-Tags UI-Klassen entsprechen. Das Data Binding

```
FontSize="{Binding LabelFontSize, ↵
                          ElementName=Page}"
```

verknüpft Eigenschaften des Bedienelements mit Variablen im Programm. Üblicherweise teilen Ereignisse den Anwendungen Reaktionen auf Nutzerinteraktionen mit. Code, den die App beispielsweise nach dem Anklicken eines Buttons ausführen soll, referenziert man mit dem Attribut `Click="Button_Click"`. Eine gleichnamige Methode befindet sich in der Datei *MainPage.xaml.cs*. Ähnliche Konzepte haben sich auch unter Android, iOS und dem Swing-Nachfolger JavaFX etabliert.

Haarkleine Änderungen, signifikante Auswirkungen

So praktisch und sinnvoll es auch ist, die Oberflächenbeschreibung und den Code zu trennen: Am grundsätzlichen Dilemma ändert das nichts. Wurde das Objektgeflecht erst einmal erzeugt, muss man es den Eingaben des Nutzers entsprechend verändern. Was dabei passieren soll, geben Entwickler haarklein vor. Ein solches Vorgehen nennt man imperativ. Bei trivialen Apps wie dem Zähler-Beispiel ist das zwar noch simpel, sobald Zustandsände-



Mit Java Swing umgesetzt, resultiert das Zähler-Beispiel in einem überschaubaren Komponentenbaum – umfangreiche Apps kann man schnell nicht mehr überblicken (Abb. 1).

Listing 1: CounterDemoSwing.java

```
@SuppressWarnings("serial")
public class CounterDemoSwing extends JFrame {

    private Font font1;
    private Font font2;

    private CounterDemoSwing() {
        super(CounterDemoSwing.class.getSimpleName());
        setContentPane(createUI());
        setSize(400, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE); }

    private JComponent createUI() {
        int[] counter = new int[] { 0 };
        var box = Box.createVerticalBox();
        box.setBorder(BorderFactory.createEmptyBorder(16, 16, 16, 16));
        var label = new JLabel();
        font1 = label.getFont().deriveFont(Font.ITALIC, 14f);
        font2 = label.getFont().deriveFont(Font.BOLD, 72f);
        var panel = new JPanel();
        panel.setAlignmentX(0.5f);
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        panel.add(Box.createVerticalGlue());
        panel.add(label);
        panel.add(Box.createVerticalGlue());
        box.add(panel);
        var button = new JButton("Klick");
        button.addActionListener(e -> updateUI(label, ++counter[0]));
        button.setAlignmentX(0.5f);
        box.add(button);
        updateUI(label, counter[0]);
        return box; }

    private void updateUI(JLabel label, int counter) {
        if (counter == 0) {
            label.setFont(font1);
            label.setText("Noch nicht geklickt");
        } else {
            label.setFont(font2);
            label.setText(Integer.toString(counter)); }}

    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new CounterDemoSwing().setVisible(true)); }}
```

Listing 2: MainPage.xaml

```
<Page x:Class="CounterDemoUWP.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    x:Name="Page"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Vertical"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Center">
        <Grid Height="100"
              HorizontalAlignment="Stretch">
            <TextBlock x:Name="Counter"
                       VerticalAlignment="Center"
                       HorizontalAlignment="Center"
                       FontWeight="{Binding ViewModel.LabelFontWeight, ElementName=Page}"
                       FontStyle="{Binding ViewModel.LabelFontStyle, ElementName=Page}"
                       FontSize="{Binding ViewModel.LabelFontSize, ElementName=Page}"
                       Text="{Binding ViewModel.Label, ElementName=Page}" />
        </Grid>
        <Button Content="Klick"
                Click="Button_Click"
                HorizontalAlignment="Center" />
    </StackPanel>
</Page>
```

Listing 3: main.dart

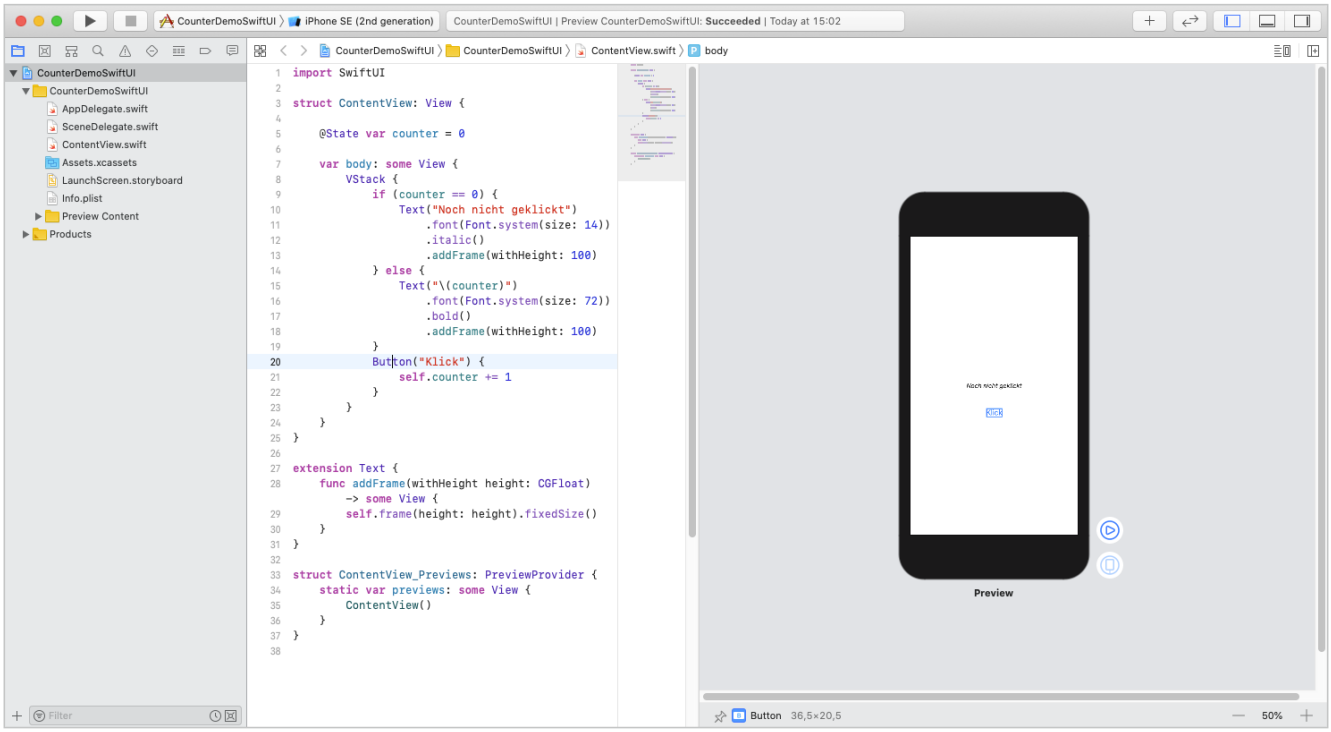
```
void main() => runApp(CounterDemo());

class CounterDemo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: _title,
      home: CounterDemoHomePage(title: _title),
    ); }}

class CounterDemoHomePage extends StatefulWidget { ,
          CounterDemoHomePage({Key key, this.title}) : super(key: key);
  final String title;

  @override
  _CounterDemoHomePageState createState() => _ ,
                                      CounterDemoHomePageState(); }

class _CounterDemoHomePageState extends State<CounterDemoHomePage> {
  int _counter = 0;

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Container(
              height: 100,
              alignment: Alignment.center,
              child: _createText(_counter),
            ),
            MaterialButton(
                textTheme: ButtonTextTheme.primary,
                child: Text("Klick"),
                onPressed: () => setState(() {
                      _counter++;
                    }))],),),); }

Widget _createText(int counter) {
  if (counter == 0) {
    return Text(
      "Noch nicht geklickt",
      style: TextStyle(fontSize: 14, fontStyle: FontStyle.italic),
      textAlign: TextAlign.center,);
  } else {
    return Text('$counter',
        style: TextStyle(
            fontSize: 72,
            fontStyle: FontStyle.normal,
            fontWeight: FontWeight.bold),
        textAlign: TextAlign.center); }}}
```

rungen aber signifikante Auswirkungen auf die Oberfläche haben, können sie Programmierer nur noch mit hohem Aufwand synchron halten. Im schlimmsten Fall entspricht die UI nicht mehr dem aktuellen Zustand.

Statt die Oberfläche auf Basis von Zustandsänderungen zu modifizieren, kann man sie situationsabhängig beschreiben. Für das Zähler-Beispiel bedeutet das:

- Die Oberfläche ist von genau einem Wert abhängig: dem aktuellen Zählerstand.
- Der Button mit dem Label Klick ist immer zu sehen.
- Haben Nutzer ihn noch nicht angeklickt, erscheint über ihm der Text „Noch nicht angeklickt“.
- Haben die Anwender ihn mindestens einmal angeklickt, erscheint stattdessen die Zahl der Klicks.

Bei deklarativ erzeugten Oberflächen spielen ausschließlich die Bedienelemente eine Rolle, die die App aktuell darstellt. Ob sie früher schon vorhanden waren und das Programm sie deshalb aktualisiert oder ob sie neu hinzukommen und die Anwendung sie deshalb in den Komponentenbaum einfügen muss, kann den Entwicklern egal sein. Dasselbe gilt, wenn Elemente wegfallen – sie interessiert nichts weiter als die augenblickliche Situation, deshalb beschreiben sie auch allein sie. Das UI-Framework muss indessen genau Buch führen, welche Änderungen nötig sind – ein vollständiges Neuzeichnen würde zu deutlichem Flackern



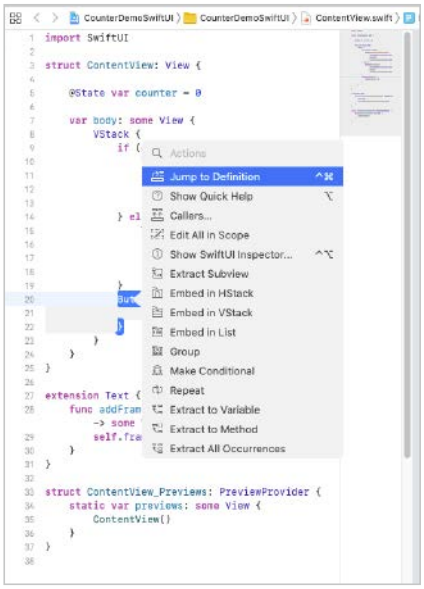
Apple setzt auf SwiftUI, mit dem sich das Zähler-Beispiel für iOS deklarativ umsetzen lässt (Abb. 2).

führen und unnötig Rechenzeit kosten. Googles Cross-Platform-Toolkit Flutter setzt voll auf deklarative UIs. Apps schreibt man in der Programmiersprache Dart. Wie man Bedienelemente kombiniert, zeigen die – mehrfach vorhandenen – Methoden `build()` und `_createText()` in Listing 3. Alle Aspekte der Benutzeroberfläche repräsentiert Flutter als Widgets. Hierzu zählen nicht nur Buttons, Labels und Grafiken, sondern auch Farben, Textauszeichnungen, Ausrichtungen und Abstände. Die UI entsteht durchs Kombinieren vieler kleiner Bausteine. Anders als bei imperativen Frameworks ist es deshalb nicht nötig, von einem Button abzuleiten und sein Verhalten zu überschreiben.

Widgets lassen sich grob in zwei Gruppen unterteilen: zustandslos und zustandsbehaftet. Text und Center gehören zur ersten Kategorie. Einmal erzeugt, lassen sie sich nicht mehr verändern. Die Methode `build()` der Klasse `StatelessWidget` und ihrer Kinder liefert ein Widget, das einen bestimmten Teil der Oberfläche repräsentiert. Im Fall der Klasse `CounterDemo` ist das `MaterialApp` – ein `StatefulWidget`. Zustandsbehaftete Widgets haben keine `build()`-Methode, stattdessen liefert ihre Methode `createState()` einen Zustand. Er leitet von der Klasse `State` ab und enthält die für das Erstellen der UI nötige Methode `build()`. Bei Zustandsänderungen ruft Flutter diese auf und zeichnet danach die Oberfläche neu. Soll sich nach Benutzeraktionen, etwa dem Anklicken eines Buttons, der State ändern, muss man nur `setState()` aufrufen. Weil die `build()`-Methode den dann gewünschten Stand der UI repräsentiert, braucht das Text-Widget keinen Setter zum Ändern seines Inhalts zu enthalten – das System erzeugt einfach ein neues Widget mit dem gewünschten Text.

SwiftUI und React

Auch Apple hat das – langsame – Ende seiner imperativen UI-Frameworks UIKit und AppKit eingeläutet. Mit SwiftUI und Xcode 11 lassen sich die Oberflächen für iOS 13, watchOS 6, tvOS 13 und macOS Catalina in Swift deklarativ erstellen. Aber noch geht es nicht ganz ohne die alte Welt: Wirft man zum Beispiel einen Blick auf SwiftUI-Projekte für iOS, findet man neben dem altbekannten `AppDelegate` einen `SceneDelegate`, der SwiftUI-Views in einen `UIHostingController` packt. Komponenten aus dem jeweils anderen Framework lassen sich direkt integrieren.



Xcode hilft beim Zusammenbauen der UI durch Kombination (Abb. 3).

Abbildung 2 zeigt den Quellcode und eine Vorschau des Zähler-Beispiels. SwiftUI basiert auf Views, wobei es sich um Strukturen handelt, die das gleichnamige Protokoll umsetzen. Views müssen mindestens einen `body` enthalten, der das Aussehen und Verhalten der View definiert. Beides lässt sich mit sogenannten Previews ausprobieren, die man mit `PreviewProvidern` erzeugt. Solcher Code ist nicht für produktive Apps gedacht und deshalb rahmt ihn `#if DEBUG ... #endif` ein.

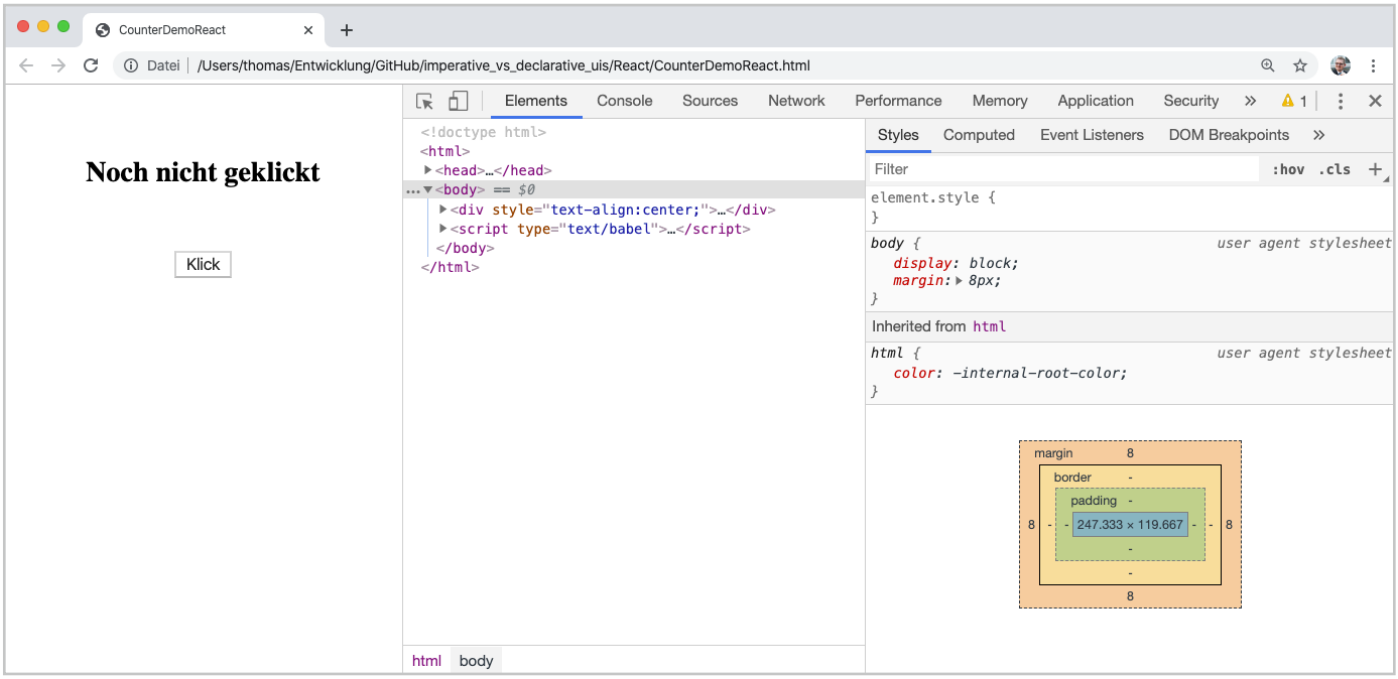
Zustände versieht man mit dem Attribut `@State`, wobei es sich um sogenannte Property Wrapper handelt: Die Instanz eines State ist nicht der Wert selbst, sondern ein Vehikel zum Lesen oder Verändern desselben. Änderungen führen automatisch zum Neuzeichnen der UI. Der explizite Aufruf einer Methode wie `setState()` in Flutter entfällt somit. Um auszuprobieren, wie sich die UI verhält, wenn eine Zustandsvariable einen bestimmten Wert hat, kann man diese einfach dem Initializer der `ContentView` im `PreviewProvider` übergeben:

```
struct ContentView_Previews: PreviewProvider {
  static var previews: some View {
    ContentView(counter: 42) }}
```

Recht ähnlich zu Flutter verläuft die Reaktion auf Button-Klicks. Entsprechenden Code übergibt man beim Erzeugen, das Erstellen der Oberfläche geschieht durch Kombination. Xcode hilft Entwicklern, indem es nach einem Klick auf eine View mit gedrückter Command-Taste ein Kontextmenü anzeigt (s. Abb. 3). Außerdem lassen sich Views aus einer Bibliothek in den Code übernehmen, auch Änderungen in der Preview zeigt die IDE im Quelltext an.

Orientiert an der Webentwicklung

Weder Apple noch Google haben deklarative UIs erfunden. Beide kombinieren erfolgreich viele im Laufe der Jahre entstan-



Das Zähler-Beispiel umgesetzt als React-Version – Webanwendungen programmieren Entwickler schon lange deklarativ (Abb. 4).

dene Ideen und Konzepte, nicht zuletzt aus der Webentwicklung. Ein frühes Beispiel ist die Trennung von Oberfläche und Code. Das eigentliche UI beschreibt man mit HTML; wie eine Überschrift letztlich aussieht, steht mit CSS an anderer Stelle. Und JavaScript ist dafür zuständig, was nach dem Anklicken eines Buttons passiert.

Moderne Webframeworks bauen auf diesem Fundament auf und machen den Browser zu einer mächtigen Laufzeitumgebung. React von Facebook beispielsweise strukturiert eine Anwendung in sogenannte Komponenten, die sich wie normale HTML-Tags verwenden lassen. Ein Zähler-Beispiel als React-Komponente hält der *iX*-Listing-Server vor. Der Einfachheit halber stecken bei dieser Umsetzung die Beschreibung der Oberfläche sowie der JavaScript-Code in einer Datei – sauberer ist es freilich, beides zu trennen. Auch das direkte Referenzieren der Bibliotheken ist nicht produktionstauglich. Andererseits lässt sich die Demo ohne weitere Entwicklungswerkzeuge einfach im Browser ausprobieren (s. Abb. 4).

React-Komponenten halten ihren Zustand in einem `state`-Objekt. Den Zähler initialisiert man mit der Anweisung `this.state = { counter: 0 };` mit 0. Um ihn beim Anklicken ums eins zu erhöhen, ist folgender Code nötig:

```
handleClicked() {
    this.setState(state => ({
        counter: state.counter + 1 })); }
```

React-Komponenten liefern ihre Oberfläche aus, indem sie die Methode `render()` implementieren. Sie erzeugt entweder HTML oder – wie das Beispiel dieses Artikels – in der an XML angelehnten Template-Sprache JSX (JavaScript Syntax Extension) formulierten Code.

Neben SwiftUI und Flutter soll auch für natives Android ein deklaratives UI-Framework geben: Google hat im Herbst 2019 bereits eine erste Vorschauversion von Jetpack Compose veröffentlicht, das sich nur mit Kotlin nutzen lässt, dafür aber optimal in die Sprache integriert ist. Noch arbeitet Google intensiv an der Fertigstellung, aber Neugierige können schon jetzt damit experimentieren. Basis für das Erstellen einer UI in Jetpack Compose sind Composable Functions. Sie sind Flutter und SwiftUI sehr ähnlich. Zustände werden in Klassen gehalten, die mit @Model annotiert sind:

```
@Model
class CounterState(var counter: Int)
@Composable
fun CountButton(state: CounterState) {
Button(
onClick = { state.counter++ }) {
Text(text = „ Klick„ )
}
}
```

Fazit

Klassische imperative Frameworks sind mächtig, denn Entwickler können die UI praktisch beliebig anpassen. Gerade in dieser Flexibilität liegt aber auch ihre größte Schwäche: je umfangreicher die UI, desto aufwendiger die Umsetzung. Deklarative Frameworks sind ebenfalls mächtig, geben Entwicklern aber mehr Struktur vor – zum Beispiel durch die Notwendigkeit, UI-Elemente ausschließlich beim Instanziieren konfigurieren zu können. Man darf davon ausgehen, dass ähnlich dem deklarativen Web schon bald deklarative App-Oberflächen dominieren. (ane@ix.de)

Quellen

Alle Listings zu diesem Artikel unter ix.de/z1e9

Thomas Künneth

arbeitet als Principal Consultant für die MATHEMA Software GmbH. Neben zahlreichen Artikeln hat er drei Bücher über Android, Java und Eclipse veröffentlicht.