

UNIVERSITÄT HEIDELBERG
SOMMERSEMESTER 2015

Softwareentwicklung für iOS

APP KATALOG

NILS FISCHER

Aktualisiert am 15. Juni 2015
Kursdetails und begleitende Materialien auf der Vorlesungswebseite:
<http://ios-dev-kurs.github.io>

Inhaltsverzeichnis

1	Über dieses Dokument	3
2	Hello World	4
2.1	Grundlagen der Programmierung in Swift	4
2.2	Objektorientiertes "Hello World!"	8
2.3	Das erste Xcode Projekt	11
2.4	"Hello World!" on Simulator	12
2.5	"Hello World!" on Device	13
2.6	Graphisches "Hello World!"	14
3	Versionskontrolle mit Git	21
4	View Hierarchie	29
4.1	Handgeschriebene View Hierarchie	29
4.2	Auto Layout	34
5	Cities	38
5.1	One City	38
5.2	One City Navigation	47
5.3	More Cities	49

Kapitel 1

Über dieses Dokument

Dieser App Katalog enthält Schritt-für-Schritt Anleitungen für die im Rahmen unseres Kurses erstellten Apps sowie die wöchentlich zu bearbeitenden Übungsaufgaben und wird im Verlauf des Semesters kapitelweise auf der Vorlesungswebseite ^[1] zur Verfügung gestellt.

Er dient jedoch nur als Ergänzung zum parallel verfügbaren **Skript**, auf das hier häufig verwiesen wird. Dort sind die Erläuterungen zu den verwendeten Technologien, Methoden und Begriffen zu finden.

¹<http://ios-dev-kurs.github.io/>

Kapitel 2





Hello World

Was ist schon ein Programmierkurs, der nicht mit einem klassischen *Hello World* Programm beginnt? Wir werden jedoch noch einen Schritt weitergehen und diesen Gruß vom iOS Simulator oder, soweit vorhanden, direkt von unseren eigenen iOS Geräten ausgeben lassen. Außerdem wird in die objektorientierte Programmierung in *Swift* eingeführt.

Relevante Kapitel im Skript: Xcode, Programmieren in Swift sowie das Buch The Swift Programming Language [1]

2.1 Grundlagen der Programmierung in Swift

Anhand des ersten Kapitels *A Swift Tour* des Buches *The Swift Programming Language* lernen wir zunächst die Grundlagen der Programmierung in Swift kennen.

1. Öffnet Xcode und erstellt zunächst einen *Playground* mit  +  +  + . Playgrounds sind interaktive Skripte, mit denen sich ideal Code ausprobieren lässt. Gebt der Datei einen Namen wie **"01 – Grundlagen der Programmierung in Swift"** und speichert sie in einem Verzeichnis für diesen Kurs.
2. Ein Playground besteht aus einem Editor- und einem Inspektorbereich und führt geschriebenen Code automatisch aus. Ausgaben und Laufzeitinformationen werden im Inspektor angezeigt. In nur einer Zeile Code können wir den traditionellen *Hello World!*-Gruß ausgeben lassen (s. S. 5, Abb. 2.1).

```
1 println("Hello World!")
```

¹https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

3. Nun lernen wir anhand des ersten Kapitels *A Swift Tour* des Buches *The Swift Programming Language* zunächst die Grundlagen der Programmierung in Swift.

Auf der Vorlesungswebseite findet ihr den Playground aus der Vorlesung, der in diese Konzepte einführt. Macht euch dabei mit folgenden Begriffen vertraut:

- Variablen (**var**) und Konstanten(**let**)
- Einfache Datentypen (**Int**, **Float**, **Double**, **Bool**, **String**, **Array**, **Dictionary** und **Set**)
- Type Inference
- String-Formatierung
- Einfache Operatoren (+, -, *, /, \%)
- Abfragen (**if**, **switch**) und Schleifen (**for**, **while**)
- Optionals
- Funktionen

Im zweiten Kapitel *Language Guide* in *The Swift Programming Language* werden diese Konzepte noch einmal detailliert erklärt. Informiert euch dort gerne genauer darüber. Zunächst genügt es jedoch, einen Überblick zu erhalten. Im Verlauf des Kurses werden wir noch viel Übung im Umgang mit diesen Konzepten bekommen.

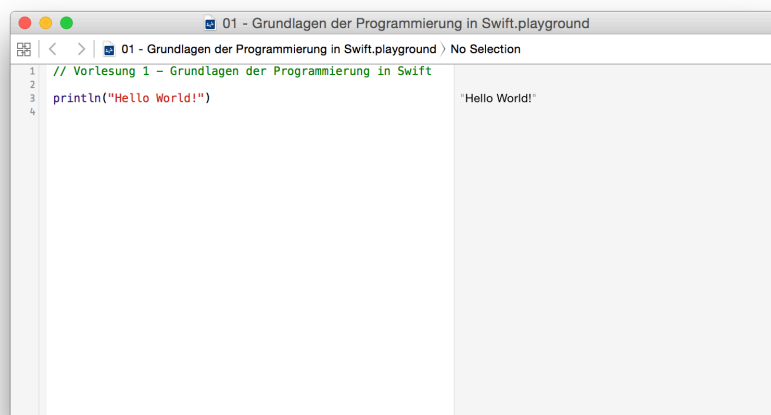


Abbildung 2.1: Playgrounds eignen sich ideal zum Ausprobieren von Swift Code.

Übungsaufgaben

1. Fibonacci [1 P.]

- a) Schreibt einen Algorithmus, der alle Folgenglieder $F_n < 1000$ der Fibonaccifolge

$$F_n = F_{n-1} + F_{n-2} \quad (2.1)$$

$$F_1 = 1, F_2 = 2 \quad (2.2)$$

in der Konsole ausgibt.

- b) **Extra:** Bei jeder geraden Fibonaccizahl F_j ist der Abstand $\Delta n = j - i$ zum vorherigen geraden Folgenglied F_i auszugeben.

2. Primzahlen [2 P.]

- a) Schreibt eine Funktion `primeNumbersUpTo:`, die ein Argument `maxNumber: Int` annimmt und alle Primzahlen bis `maxNumber` als Liste `[Int]` zurückgibt.

Hinweis: Mit dem Modulo-Operator `%` kann der Rest der Division zweier Integer gefunden werden:

```
1 let a = 20%3 // a ist jetzt Int(2)
```

- b) *Optionals* sind eines der elegantesten Konzepte in Swift, und sind auch in anderen modernen Sprachen zu finden. Informiert euch darüber im Kapitel *Language Guide > The Basics > Optionals* in *The Swift Programming Language*. Dieses Kapitel (bis einschließlich *Implicitly Unwrapped Optionals*) ist sehr wichtig, da wir in der iOS App Programmierung häufig mit Optionals arbeiten werden!

- c) Verwendet eure Liste von Primzahlen aus der vorigen Aufgabe, um effizienter zu prüfen, ob eine Zahl eine Primzahl ist.

Schreibt dazu eine Funktion `isPrimeNumber:cachedPrimeNumbers:`, die eine Zahl `n: Int` und eine **optionale** Liste von Primzahlen `cachedPrimeNumbers: [Int]?` annimmt. Verwendet die *Optional Binding Syntax* `if let` um mit dieser Liste zu arbeiten, wenn eine solche übergeben wurde und lang genug ist. Dann genügt es zu prüfen, ob die Zahl in der Liste enthalten ist. Wenn keine Liste übergeben wurde, soll die Primzahl wie in a) manuell geprüft werden.

Hinweise:

- Dem Argument `cachedPrimeNumbers` können wir einen *default* Wert `nil` zuweisen:

```
1 func isPrimeNumber(n: Int, cachedPrimeNumbers: [Int]? = nil) ->
  ↪ Bool {
2     // ...
3 }
```

So kann die Funktion auch ohne dieses Argument aufgerufen werden:

```

1 isPrimeNumber(7, cachedPrimeNumbers: [ 1, 2, 3, 7 ]) //
  ↳ vollständiger Funktionsaufruf, verwendet übergebene Liste zum
  ↳ Nachschlagen
2 isPrimeNumber(7) // äquivalent zu:
3 isPrimeNumber(7, cachedPrimeNumbers: nil) // Prüft Primzahl
  ↳ manuell

```

- Die globale Funktion `contains` prüft ob eine Element in einer Liste enthalten ist:

```

1 contains([ 1, 2, 3, 7 ], 7) // true

```

- Testet eure Funktion, indem Ihr bspw. folgenden Code ans Ende des Storyboards setzt:

```

1 //: ## Testing
2
3 let n = 499 // Number to test
4 let cachedMax = 500 // Prime numbers up to this number will be
  ↳ cached
5 import Foundation
6 var startDate: NSDate
7
8 startDate = NSDate()
9 let cachedPrimeNumbers = primeNumbersUpTo(cachedMax)
10 println("Time for caching prime numbers up to \(cachedMax):
  ↳ \(-startDate.timeIntervalSinceNow)s")
11
12 startDate = NSDate()
13 if isPrimeNumber(n) {
14     println("\(n) is a prime number.")
15 } else {
16     println("\(n) is not a prime number.")
17 }
18 let withoutCacheTime = -startDate.timeIntervalSinceNow
19 println("Time without cache: \(withoutCacheTime)s")
20
21 startDate = NSDate()
22 isPrimeNumber(n, cachedPrimeNumbers: cachedPrimeNumbers)
23 let withCacheTime = -startDate.timeIntervalSinceNow
24 println("Time with cache: \(withCacheTime)s (\((1 - withCacheTime
  ↳ / withoutCacheTime) * 100)% faster)")

```

2.2 Objektorientiertes "Hello World!"

1. Nun versuchen wir uns an der objektorientierten Programmierung und möchten den Hello World! Gruß von virtuellen Repräsentationen einzelner Personen ausgeben lassen. Erstellt dazu einen Xcode Playground bspw. mit Titel "**02 – Objektorientierte Programmierung in Swift**".
2. Verschafft euch anhand *The Swift Programming Language* und dem Playground aus der Vorlesung (auf der Vorlesungswebseite) einen Überblick über folgende Konzepte der objektorientierten Programmierung in Swift:
 - Klassen und Objekte
 - Attribute mit oder ohne Startwert
 - Initializer
 - Instanz- und Klassenmethoden
 - Subklassen und Überschreiben von Methoden
 - Structs und Enums

Übungsaufgaben

3. Scientists

[1 P.]

- a) Erstellt (am besten in einem neuen Playground, in den ihr die Klasse `Person` aus der Vorlesung einfügt) eine weitere Klasse `Scientist` als *Subklasse* von `Person`.

Wissenschaftler können rechnen, fügt dieser Klasse also eine Methode `sayPrimeNumbersUpTo:` hinzu, die ein Argument `maxNumber: Int` annimmt und alle Primzahlen bis zu dieser Zahl in der Konsole ausgibt. Verwendet dazu den Algorithmus aus der vorherigen Übungsaufgabe (s. S. 6, Übungsaufgabe 2).

Hinweis: Wie in *The Swift Programming Language* beschrieben, erbt eine Subklasse die Attribute und Methoden ihrer Superklasse und kann diese überschreiben:

```
1 class Scientist: Person {  
2     ...  
3 }
```

- b) Wir wollen uns vergewissern, dass die Klasse `Scientist` die Attribute und Methoden ihrer Superklasse `Person` erbt. Erstellt ein `Scientist`-Objekt, gebt ihm einen Namen und lasst den Hello World-Gruß ausgeben.

- c) Nach dem Prinzip des *Überschreiben* soll ein Wissenschaftler einen anderen Gruß ausgeben als eine "normale" Person. Überschreibt in der `Scientist`-Klasse die Methode `sayHello`, sodass zusätzlich **"Ask me for prime numbers!"** ausgegeben wird.

4. Poker

[3 P.]

In dieser Aufgabe berechnen wir die Wahrscheinlichkeit für einen *Flush* beim Poker.

- a) Zunächst modellieren wir die Spielkarten. Eine Karte hat immer eine *Farbe/Suit* (*Karo/Diamonds*, *Herz/Hearts*, *Pik/Spades* oder *Kreuz/Clubs*) und einen *Rang/Rank* (2 bis 10, *Bube/Jack*, *Dame/Queen*, *König/King* oder *Ass/Ace*).

Schreibt zwei Enums `enum Suit: Int` und `enum Rank: Int` mit ihren entsprechenden Fällen (`case Diamonds` usw.). Bei den Rängen 2 bis 10 schreibt ihr am besten die Zahl aus. Implementiert jeweils eine *Computed Property* `var symbol: String`, in der ihr mithilfe einer `switch`-Abfrage für jeden Fall ein Symbol zurückgibt. **Tipp:** Für die Farben gibt es Unicode-Symbole^[2]!

Schreibt dann einen `struct Card` mit zwei Attributen `let suit: Suit` und `let rank: Rank`, sowie einer *Computed Property* `var description: String`, die einen aus Farbe und Rang zusammengesetzten String zurückgibt.

- b) Nun können wir eine Poker Hand modellieren. Schreibt den `struct Poker-Hand` mit einem Attribut `let cards: [Card]` und einer *Computed Property* `var description: String`, die die description der Karten kombiniert.

Um einfach zufällige Poker Hände generieren zu können, implementiert einen Initializer `init()`, der eine Hand aus fünf zufälligen Karten erstellt. **Wichtig:** Da aus einem Deck von paarweise verschiedenen Karten gezogen wird, darf keine Karte doppelt vorkommen.

Hinweise:

- Da wir `Suit` und `Rank` von `Int` abgeleitet haben, können wir Zufallszahlen generieren und die Enums daraus erstellen:

```
1 let rndSuit = Suit(rawValue: Int(arc4random_uniform(4)))!
2 let rndRank = Rank(rawValue: Int(arc4random_uniform(13)))!
3 let rndCard = Card(suit: rndSuit, rank: rndRank) // Eine
  ↪ zufällige Spielkarte
```

- Die Funktion `contains` könnte hilfreich sein, um das Vorhandensein von Karten zu überprüfen. Um diese mit `Card` verwenden zu können, müsst ihr erst eine Äquivalenzrelation implementieren: Schreibt `struct Card: Equatable { ... }` und dann außerhalb des Struct:

²http://en.wikipedia.org/wiki/Playing_cards_in_Unicode

```

1 func ==(lhs: Card, rhs: Card) -> Bool {
2     return lhs.suit == rhs.suit && lhs.rank == rhs.rank
3 }

```

- c) Erstellt ein paar Poker Hände und lasst euch die description ausgeben. Habt ihr etwas gutes gezogen?

Implementiert nun ein weiteres Enum `enum Ranking: Int` mit den Fällen `case HighCard, Flush, StraightFlush` usw., die ihr bspw. auf Wikipedia^[3] findet.

Fügt dann dem `struct PokerHand` eine Computed Property `var ranking: Ranking` hinzu. Implementiert hier einen Algorithmus, der prüft, ob ein `Flush` vorliegt. Dann soll `.Flush` zurückgegeben werden, ansonsten einfach `.HighCard`.

- d) Wir können nun einige tausend Hände generieren und die Wahrscheinlichkeit für einen Flush abschätzen. Fügt einfach folgenden Code am Ende des Playgrounds ein:

```

1 var rankingCounts = [Ranking : Int]()
2 let samples = 1000
3 for var i=0; i<samples; i++ {
4     let ranking = PokerHand().ranking
5     if rankingCounts[ranking] == nil {
6         rankingCounts[ranking] = 1
7     } else {
8         rankingCounts[ranking]!++
9     }
10 }
11 for (ranking, count) in rankingCounts {
12     println("The probability of being dealt a \(ranking.description)
13         ↪ is \(Double(count) / Double(samples) * 100)%")
14 }

```

Die Ausführung kann etwas dauern, justiert ggfs. `samples`. Stimmt die Wahrscheinlichkeit etwa mit der Angabe auf Wikipedia überein?

- e) **Extra:** Ihr könnt das Programm nun noch erweitern und versuchen, die anderen Ränge zu überprüfen. Dabei könnten Hilfsattribute wie `var hasFlush: Bool` oder `var pairCount: Int` nützlich sein. Bekommt es jemand es jemand hin, eine Funktion zu schreiben, die zwei Hände vergleicht und den Sieger bestimmt? **Tipp:** Dazu könnte es hilfreich sein, die Fälle des `enum: Ranking` um *Associated Attributes* zu erweitern.

³http://en.wikipedia.org/wiki/List_of_poker_hands

2.3 Das erste Xcode Projekt

iOS Apps schreiben wir natürlich nicht in Playgrounds. Eine App besteht aus vielen Komponenten wie Klassen, Interface-Dateien und weiteren Ressourcen, die wir in einem Xcode Projekt zusammenfassen.

Relevante Kapitel im Skript: Xcode

1. Um nun unsere erste iOS App zu erstellen, rufen wir mit $\text{⌘} + \text{⬆} + \text{N}$ zunächst den Dialog zur Erstellung eines neuen Projekts auf und wählen das Template **iOS Application** **>> Single View Application**.
2. Tragt im erscheinenden Konfigurationsdialog entsprechend der Konventionen den Product Name **"HelloWorld"**, euren Vor- und Nachnamen als Organization Name und **"de.uni-hd.<deinname>"** als Company Identifier ein (s. S. 11, Abb. 2.2). Das führt zu der Bundle ID **"de.uni-hd.<deinname>.HelloWorld"**. Wählt **Swift**, **Universal** und entfernt die Auswahl von **Use Core Data**. Speichert das Projekt in einem Verzeichnis eurer Wahl.

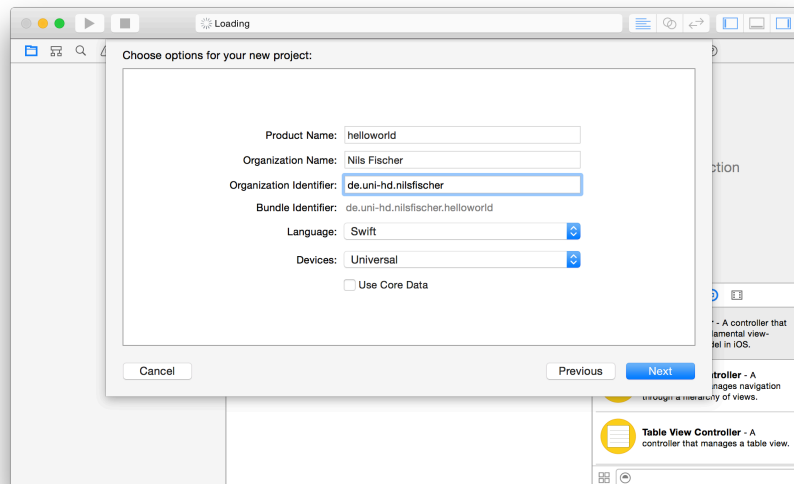


Abbildung 2.2: Damit es keine Konflikte zwischen verschiedenen Apps gibt, gibt es Konventionen bei der Konfiguration

3. Wir sehen nun Xcodes Benutzeroberfläche und können sie mit den Schaltflächen rechts in der Toolbar anpassen. Verwendet zunächst die Konfiguration mit eingeblen-detem Navigator, verstecktem Debug-Bereich und Inspektor und Standard-Editor. Wählt im Project Navigator das Projekt selbst aus (s. S. 12, Abb. 2.3).
4. Im Editor wird die *Projekt- und Targetkonfiguration* angezeigt. Hier können wir bspw. die Bundle ID unserer App anpassen, die wir zuvor bei der Erstellung des Projekts aus Product Name und Company Identifier zusammengesetzt haben.

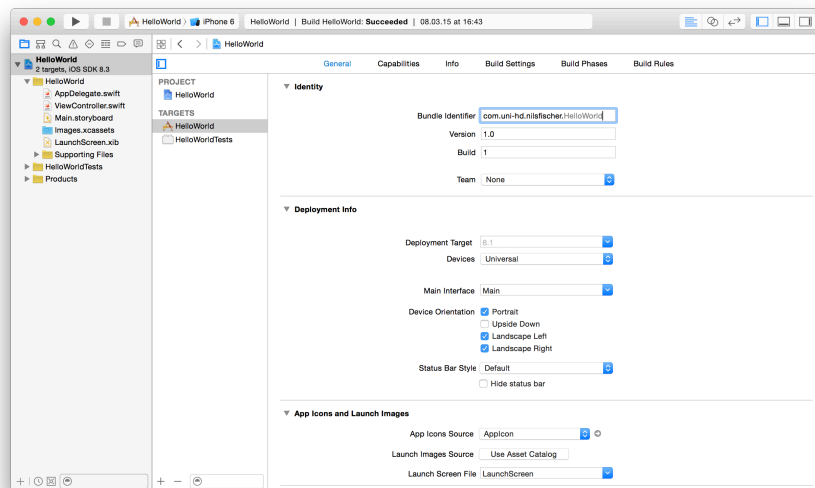


Abbildung 2.3: Wird das Projekt ausgewählt, sehen wir im Editor die Projekt- und Targetkonfiguration.

- Links in der Toolbar sind die Steuerelemente des Compilers zu finden. Wählt das gerade erstellte Target und ein Zielsystem aus, bspw. den *iPhone 6* Simulator, und klickt die *Build & Run* Schaltfläche. Das Target wird nun kompiliert und generiert ein *Product*, also unserer App, die im Simulator ausgeführt wird. In Xcode kann mit + die Ausführung gestoppt und mit + (Tastenkürzel für *Build & Run*) erneut gestartet werden.

2.4 "Hello World!" on Simulator

- Besonders spannend ist diese App natürlich noch nicht. Das ändern wir jetzt spektakulär, indem wir eine Ausgabe hinzufügen. Wählt die Datei *AppDelegate.swift* im Project Navigator aus.
- Die Methode `application:didFinishLaunchingWithOptions:` wird zu Beginn der Ausführung der App aufgerufen. Ersetzt den Kommentar dort mit dem bekannten Gruß zur Ausgabe in der Konsole:

```

1 func application(application: UIApplication,
  ↳ didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
  ↳ -> Bool {
2     println("Hello World!")
3     return true
4 }

```

3. Wenn wir unsere App nun erneut mit *Build & Run* kompilieren und ausführen, sehen wir den Text **"Hello World!"** in der Konsole. Dazu wird der zweigeteilte Debug-Bereich unten automatisch eingeblendet (s. S. 13, Abb. 2.4). Ist der Konsolenbereich zunächst versteckt, kann er mit der Schaltfläche in der rechten unteren Ecke angezeigt werden. Außerdem wird links automatisch zum Debug Navigator gewechselt, wenn eine App ausgeführt wird, in dem CPU- und Speicherauslastung überwacht werden können und Fehler und Warnungen angezeigt werden, wenn welche auftreten.

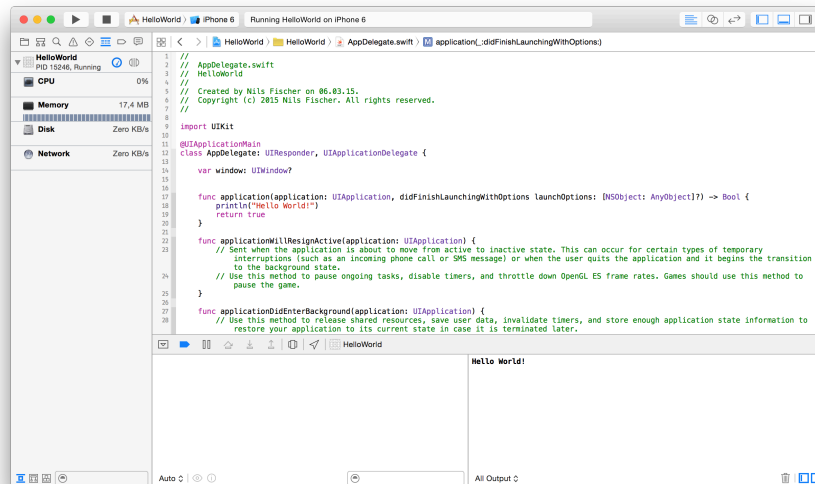


Abbildung 2.4: In der Konsole des Debug-Bereichs werden Ausgaben der laufenden App angezeigt

2.5 "Hello World!" on Device

1. Nun möchten wir unsere neue App natürlich auch auf einem realen iOS Gerät anstatt des Simulators testen. Im Skript findet ihr eine Anleitung, wie ihr mit euren iOS Geräten unserem Developer Team der Uni Heidelberg beitreten könnt.
2. Habt ihr die Schritte befolgt und euren freigeschalteten Apple Developer Account in den Xcode-Accounteinstellungen hinzugefügt, öffnet ihr wieder die Projekt- und Targetkonfiguration im Project Navigator und wählt dort unser Developer Team (s. S. 14, Abb. 2.5) aus. Nun wird automatisch das richtige Provisioning Profile für die Bundle ID des Targets verwendet.
3. Verbindet euer iOS Gerät mit eurem Mac und wählt es in der Toolbar als Zielsystem aus. Mit einem *Build & Run* wird die App nun kompiliert, auf dem Gerät installiert und ausgeführt. In der Konsole erscheint wieder die Ausgabe **"Hello World!"**, diesmal direkt vom Gerät ausgegeben.

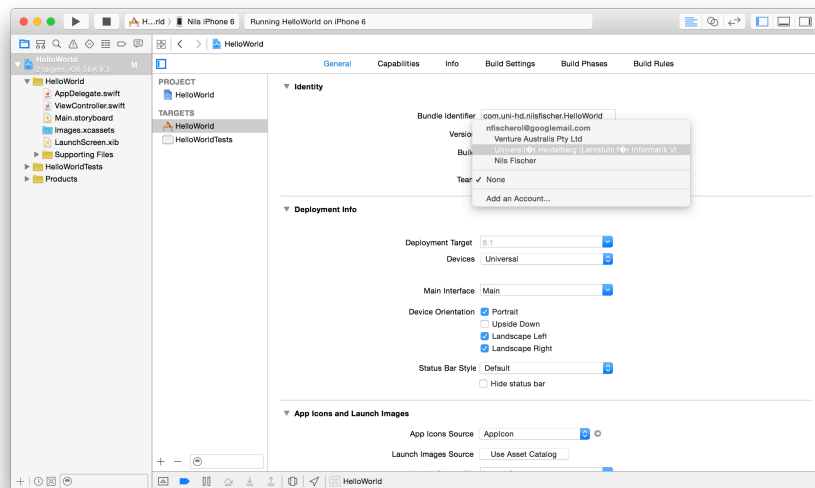




Abbildung 2.5: Mit der Wahl des zugehörigen Developer Teams in der Project- und Targetkonfiguration verwendet Xcode automatisch das passende Provisioning Profile

2.6 Graphisches "Hello World!"

Natürlich wird ein Benutzer unserer App von den Ausgaben in der Konsole nichts mitbekommen. Diese dienen bei der Programmierung hauptsächlich dazu, Abläufe im Code nachzuvollziehen und Fehler zu finden. Unsere App ist also nur sinnvoll, wenn wir die Ausgaben auch auf dem Bildschirm darstellen können.

Relevante Kapitel im Skript: Xcode

1. Zur Gestaltung der Benutzeroberfläche oder *User Interface (UI)* verwenden wir den in Xcode integrierten *Interface Builder (IB)*. Wir haben bei der Projekterstellung der ersten App das *Single View*-Template ausgewählt, daher enthält das Projekt bereits ein *Storyboard*. Öffnet das Projekt und wählt im Navigator die Datei *Main.storyboard* aus.
2. Der Editor-Bereich zeigt nun den Interface Builder. In diesem Modus möchten wir häufig eine angepasste Konfiguration des Xcode-Fensters verwenden, es bietet sich also an, mit  +  einen neuen Tab zu öffnen. Blendet dann mit den Schaltflächen in der Toolbar den Navigator- und Debug-Bereich aus und den Inspektor ein. Wählt dort außerdem zunächst den Standard-Editor (s. S. 15, Abb. 2.6).
3. Unser UI besteht bisher nur aus einer einzigen Ansicht, oder *Scene*. Ein Pfeil kennzeichnet die Scene, die zum Start der App angezeigt wird. Im Inspektor ist unten die *Object Library* zu finden. Wählt den entsprechenden Tab aus, wenn er noch nicht angezeigt wird (s. S. 15, Abb. 2.6).
4. Durchsucht die Liste von Interfaceelementen nach einem Objekt der Klasse `UILabel`, indem ihr das Suchfeld unten verwendet, und zieht ein Label irgendwo auf die erste Scene. Doppelklickt auf das erstellte Label und tippt "Hello World!".

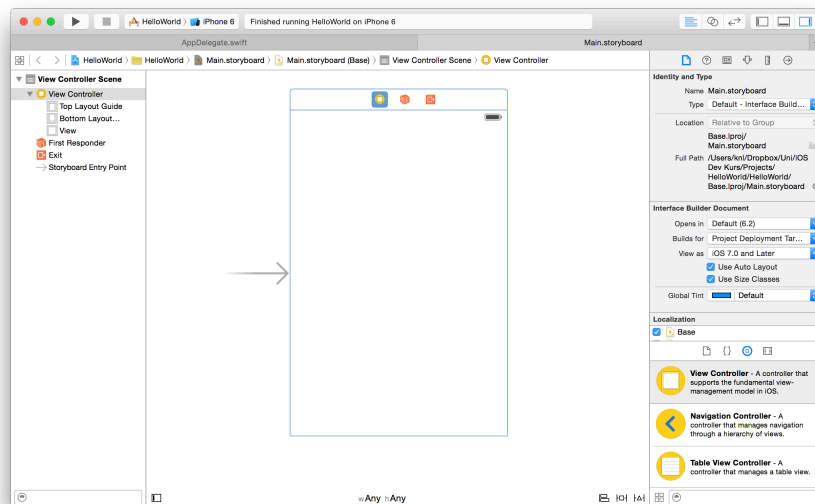


Abbildung 2.6: Für den Interface Builder verwenden wir eine angepasste Fensterkonfiguration mit dem Inspektor anstatt des Navigators

5. Ein *Build & Run* mit einem iPhone-Zielsystem zeigt diesen Gruß nun statisch auf dem Bildschirm an.
6. Habt ihr das Label im Interface Builder ausgewählt, zeigt der Inspektor Informationen darüber an. Im *Identity Inspector* könnt ihr euch vergewissern, dass das Objekt, was zur Laufzeit erzeugt wird und das Label darstellt, ein Objekt der Klasse `UILabel` ist. Im *Attributes Inspector* stehen viele Optionen zur Auswahl, mit denen Eigenschaften wie Inhalt, Schrift und Farbe des Labels angepasst werden können.
7. Natürlich möchten wir unser UI zur Laufzeit mit Inhalt füllen und den Benutzer mit den Interfaceelementen interagieren lassen können. Zieht ein `UIButton`- und `UITextField`-Objekt auf die Scene und positioniert sie passend (s. S. 16, Abb. 2.7). Mit dem Attributes Inspector könnt ihr dem Button nun den Titel "Say Hello!" geben und für das Text Field einen Placeholder "Name" einstellen.
8. Zur Laufzeit der App wird für jedes im Storyboard konfigurierte Interfaceelement ein Objekt der entsprechenden Klasse erstellt und dessen Attribute gesetzt. Um nun im Code auf die erstellten Objekte zugreifen und auf Benutzereingaben reagieren zu können, verwenden wir *IBOutlets* und *IBActions*.

Blendet den Inspektor aus und wählt stattdessen den Assistant-Editor in der Toolbar. Stellt den Modus in der Jump bar auf *Automatic*. Im Assistant wird automatisch die Implementierung des übergeordneten View Controllers eingeblendet (s. S. 17, Abb. 2.8).

9. *View Controller* sind Objekte einer Subklasse von `UIViewController`, die jeweils einen Teil der App steuern. Diese sind zentrale Bestandteile einer App, mit denen wir uns noch detailliert beschäftigen werden. Ein erster View Controller zur Steuerung dieser ersten Ansicht wurde bereits bei der Projekterstellung automatisch hinzugefügt.

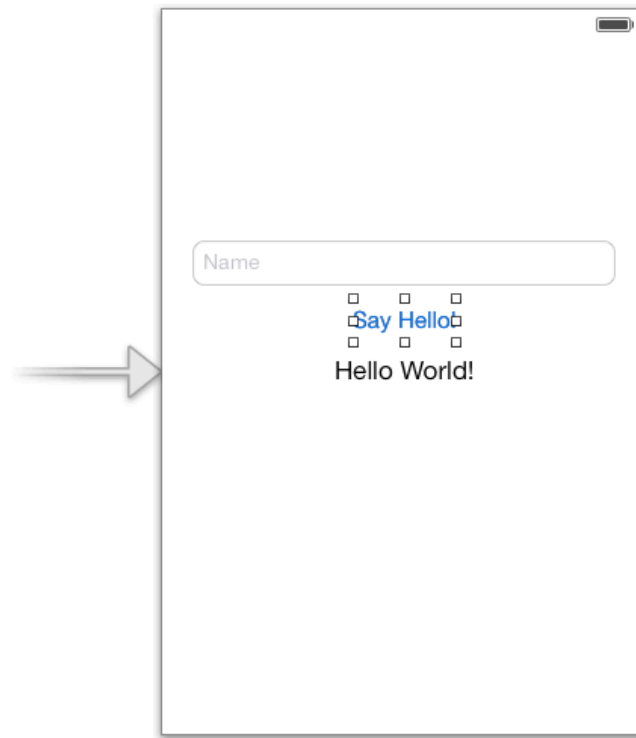


Abbildung 2.7: Mit einem Text Field, einem Button und einem Label erstellen wir ein simples UI

Fügt dieser Klasse ViewController: `UIViewController` Attribute für das `UILabel` und das `UITextField` hinzu und kennzeichnet diese mit `@IBOutlet`. Implementiert außerdem eine mit `IBAction` gekennzeichnete Methode, die aufgerufen werden soll, wenn der Benutzer den `UIButton` betätigt:

```
1 import UIKit
2
3 class ViewController: UIViewController {
4
5     @IBOutlet var nameTextField: UITextField!
6     @IBOutlet var greetingLabel: UILabel!
7
8     @IBAction func greetingButtonPressed(sender: UIButton) {
9         println("Hello World!")
10    }
11
12 }
13 @end
```

Beachtet, dass IBOutlets im Allgemeinen als *Implicitly Unwrapped Optionals* deklariert

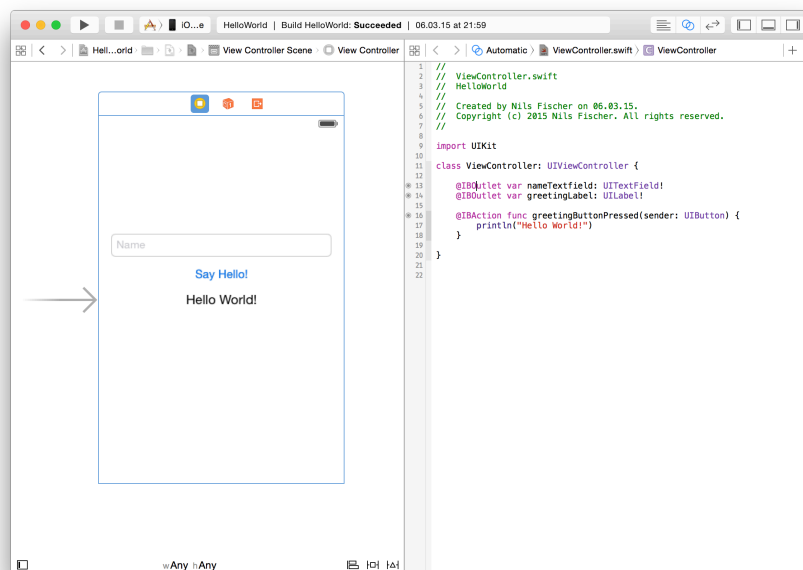


Abbildung 2.8: Mithilfe des Assistants können Interface-BUILDER und Code nebeneinander angezeigt werden.

- werden, da ihr Wert bei der Initialisierung des View Controllers zwar noch nicht existiert, anschließend jedoch unter Verwendung des Storyboards zuverlässig zugewiesen wird. So muss zwar unbedingt aufgepasst werden, dass die IBOutlet Verbindung im Storyboard hergestellt ist und nicht auf das Attribut zugegriffen wird, bevor dieses geladen wurde, doch es ist dann nicht jedes mal notwendig, das Optional zu entpacken.
10. Nun zieht mit gedrückter `ctrl`-Taste eine Linie von dem Textfeld und dem Label im Interface Builder auf das jeweilige Attribut im Code. Die Codezeile wird dabei blau hinterlegt. Zieht außerdem genauso eine Linie von dem Button auf die zuvor definierte Methode. Im Connection Inspector könnt ihr die IBOutlets und IBActions eines ausgewählten Objekts betrachten und wieder entfernen. Dieser Prozess ist im Skript noch detaillierter beschrieben.
 11. Versucht nun einen *Build & Run*. Betätigt ihr den Button, wird die Methode ausgeführt und der Gruß "Hello World!" in der Konsole ausgegeben!
 12. Um die App nun alltagstauglich zu gestalten, muss dieser Gruß natürlich personalisiert und auf dem Bildschirm angezeigt werden. Dazu verwenden wir das Attribut text der Klassen `UITextField` und `UILabel`:

```

1 @IBAction func greetingButtonPressed(sender: UIButton) {
2     self.greetingLabel.text = "Hello \(self.nameTextfield.text)!"
3 }

```

Nach einem *Build & Run* erhalten wir unser erstes interaktives Interface, in dem ihr im Textfeld einen Namen eintippen könnt und persönlich begrüßt werdet (s. S. 18, Abb. 2.9)!

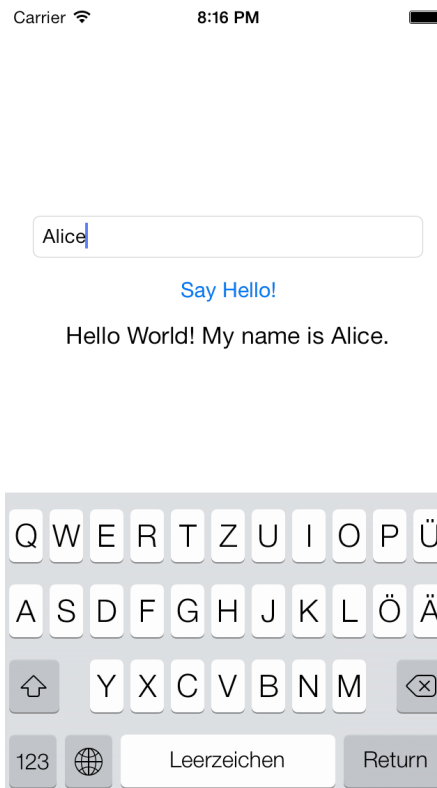


Abbildung 2.9: Drücken wir auf den Button, werden wir persönlich begrüßt. Sehr praktisch!

Übungsaufgaben

5. Simple UI [2 P.]

Erstellt ein neues Projekt und schreibt eine App mit einigen Interfaceelementen, die etwas sinnvolles tut.

Implementiert eines der folgenden Beispiele oder eine eigene Idee. Ich freue mich auf kreative Apps!

Counter Auf dem Bildschirm ist ein Label zu sehen, das den Wert eines Attributs `var count: Int` anzeigt, wenn eine Methode `updateLabel` aufgerufen wird. Buttons mit den Titeln `" +1 "`, `" -1 "` und `"Reset"` ändern den Wert dieses Attributs entsprechend und rufen die `updateLabel`-Methode auf.

BMI Nach Eingabe von Gewicht m und Größe l wird der Body-Mass-Index^[4] $BMI = m/l^2$ berechnet und angezeigt. Als Erweiterung kann die altersabhängige Einordnung in die Gewichtskategorien angezeigt werden.

⁴<http://de.wikipedia.org/wiki/Body-Mass-Index>

RGB In drei Textfelder kann jeweils ein Wert zwischen 0 und 255 für die Rot-, Grün- oder Blau-Komponenten eingegeben werden. Ein Button setzt die Hintergrundfarbe `self.view.backgroundColor` entsprechend und ein weiterer Button generiert eine zufällige Hintergrundfarbe. Ihr könnt noch einen `UISwitch` hinzufügen, der einen Timer ein- und ausschaltet und damit die Hintergrundfarbe bei jedem Timerintervall zufällig wechselt (s. Hinweis).

Hinweise:

- Achtet darauf, dass ihr bei/nach der Projekterstellung in der Targetkonfiguration die Bundle ID `"de.uni-hd.<deinname>.<productname>"` mit `"<productname>"` z.B. `"Counter"`, `"BMI"` oder `"RGB"` eingestellt und unser Developer Team ausgewählt habt, damit die Ausführung der App auf euren eigenen Geräten funktioniert!
- Die Klasse `NSString` aus Apple's Foundation Framework besitzt Instanzmethoden wie `floatValue` zur Umwandlung von Text in Zahlenwerte. `String` und `NSString` sind direkt ineinander überführbar:

```
1 let s = "0.1"
2 let f: Float = (s as NSString).floatValue
```

Eleganter ist die Verwendung eines `NSNumberFormatter`. Solche Formatter, wie auch der `NSDateFormatter`, berücksichtigen bspw. Gerätesprache und länderspezifische Einstellungen und sollten stets verwendet werden, wenn Benutzereingaben interpretiert oder Ausgaben generiert werden:

```
1 let decimalNumberFormatter = NSNumberFormatter()
2 decimalNumberFormatter.numberStyle = .DecimalStyle
3 let s: String = "0.1"
4 let f: Float? =
    ↪ decimalNumberFormatter.numberFromString(s)?.floatValue
```

- Der Initializer `UIColor(red:green:blue:alpha:)` von `UIColor` akzeptiert jeweils Werte zwischen 0 und 1.
- Die Funktion `arc4random_uniform(n)` gibt eine Pseudozufallszahl x mit $0 \leq x < n$ aus.
- Wenn ein `UISwitch` betätigt wird, sendet dieser ein Event `UIControlEvent.ValueChanged`, so wie ein `UIButton` das Event `UIControlEvent.TouchUpInside` sendet. Dieses Event kann genauso mit einer `IBAction` verbunden werden. Mit einem Attribut `var randomTimer: NSTimer?` können wir dann die Methode für das zufällige Wechseln der Hintergrundfarbe implementieren:

```
1 @IBAction func switchValueChanged(sender: UISwitch) {
2     if sender.on {
```

```
3         self.randomTimer =  
           ↳ NSTimer.scheduledTimerWithTimeInterval(0.15, target:  
           ↳ self, selector: "randomButtonPressed:", userInfo: nil,  
           ↳ repeats: true)  
4     } else {  
5         self.randomTimer?.invalidate()  
6         self.randomTimer = nil  
7     }  
8 }
```

Somit wird periodisch die Methode `randomButtonPressed:` aufgerufen, die natürlich implementiert sein muss.

Kapitel 3

Versionskontrolle mit Git

Im Skript sind die Vorzüge der Versionskontrolle mit Git beschrieben, deren Grundlagen wir anhand einer unserer Apps anwenden können.

Relevante Kapitel im Skript: Versionskontrolle mit Git

1. Da Git in erster Linie ein Kommandozeilenprogramm ist, verwenden wir zunächst die Konsole. Später könnt ihr stattdessen auch bspw. die in Xcode integrierten Benutzeroberflächen verwenden. Öffnet die *Terminal* App und navigiert in den Ordner, der das Xcode Projekt der *Hello World* App beinhaltet. Dazu kann es hilfreich sein, zunächst `cmd` zu tippen und den Ordner dann auf das Terminal Fenster zu ziehen, wobei der Pfad automatisch eingegeben wird.

```
1 cd path/to/project
```

2. Bei Erstellung des Projektes wurde möglicherweise bereits die Option ausgewählt, ein Git Repository zu initialisieren. Mit `git status` können wir prüfen, ob hier bereits eines existiert und es ansonsten mit `git init` erstellen.

```
1 git status
2 # Git Repository vorhanden:
3 >> On branch master
4 >> nothing to commit, working directory clean
5 # kein Git Repository vorhanden:
6 >> fatal: Not a git repository (or any of the parent directories): .git
7 git init
```

3. Wir fügen dem Repository nun zunächst eine `.gitignore` Datei hinzu, um verschiedene benutzerspezifische und temporäre Dateien des Xcode Projekts auszuschließen.

```
1 touch .gitignore
2 open .gitignore
```

Kopiert die Vorlagen *Xcode* ^[1] und *OSX* ^[2] in die `.gitignore` Datei.

Nun können wir einen Commit ausführen, um die Datei dem Repository hinzuzufügen.

```
1 git add .gitignore
2 git commit -m "Added .gitignore file"
```

4. Jederzeit kann es hilfreich sein, die Situation des Git Repositories mit `git status` zu überprüfen. Zeigt ein Aufruf dieses Befehls noch ungesicherte Änderungen an, könnt ihr diese in einem weiteren Commit sichern:

```
1 git add --all
2 git commit -m "Committed unsaved changes"
```

`git log` zeigt die letzten Commits in der Konsole an.

5. Nun können wir an unserem Projekt weiterarbeiten und Änderungen an Dateien vornehmen. Öffnet bspw. die Datei *ViewController.swift* und fügt ihrer Implementierung folgendes Codesegment hinzu:

```
1 override func viewDidLoad() {
2     super.viewDidLoad()
3     self.view.backgroundColor = UIColor.redColor()
4 }
```

Führt ihr die App nun aus, seht ihr einen roten Bildschirmhintergrund.

In der Konsole sehen wir mit `git status`, dass nun Änderungen vorliegen. Diese können wir in Form eines Commits im Git Repository speichern.

```
1 git add ViewController.swift # oder git add --all
2 git commit -m "changed background color"
```

¹<https://github.com/github/gitignore/blob/master/Global/Xcode.gitignore>

²<https://github.com/github/gitignore/blob/master/Global/OSX.gitignore>

6. Es gibt viele verschiedene Möglichkeit der Commitnavigation und -manipulation. Wir können bspw. mit `git checkout` einen bestimmten Commit laden, mit `git reset` Commits entfernen oder sie mit `git revert` in Form eines neuen Commits rückgängig machen. Dabei können wir einen bestimmten Commit anhand seines SHA hashes identifizieren, der bspw. mit `git log` angezeigt wird, oder mit `HEAD~x` den x-letzten Commit auswählen. Setzen wir den soeben ausgeführten Commit nun also bspw. zurück:

```
1 git reset --hard HEAD~1 # Verwendet diesen Befehl nicht leichtfertig,  
  ↳ denn hier gibt es keine Undo-Funtion!
```

In der Dokumentation ^[3] kann sich ausführlich über die verschiedenen Möglichkeiten informiert werden.

7. Häufig wird Git zur Projektstrukturierung in Form von **Feature Branches** eingesetzt. Nehmen wir also an unser Projekt liegt in seiner veröffentlichten Form vor. Möchten wir nun ein neues Feature implementieren oder Umstrukturierungen vornehmen, erstellen wir zunächst einen neuen Branch mit `git branch`. Mit `git checkout` wechseln wir das Arbeitsverzeichnis in diesen neuen Branch.

```
1 git branch new_feature  
2 git checkout new_feature
```

8. Nun können wir in diesem Branch an unserem Projekt arbeiten, ohne andere Branches wie den zuvor verwendeten master Branch zu verändern. Implementiert wieder eine Änderung und führt einen Commit durch:

```
1 override func viewDidLoad(animated: Bool) {  
2     super.viewDidLoad()  
3     let alertController = UIAlertController(title: "New Feature!",  
4     ↳ message: "Is it a bug or a feature?", preferredStyle: .Alert)  
5     alertController.addAction(UIAlertAction(title: "Bug", style:  
6     ↳ .Destructive, handler: nil))  
7     alertController.addAction(UIAlertAction(title: "Feature", style:  
8     ↳ .Default, handler: nil))  
9     self.presentViewController(alertController, animated: animated,  
10    ↳ completion: nil)  
11 }
```

```
1 git commit -a -m "implemented new feature" # Der -a Flag fügt dem Commit  
  ↳ automatisch alle veränderten Dateien hinzu
```

³<http://git-scm.com/book/>

9. Erhalten wir nun plötzlich eine Email eines aufgebrachten Benutzers unserer App, der einen Fehler gefunden hat, können wir problemlos zurück zum master Branch wechseln und diesen schnell beheben:

```
1 git checkout master
```

Nachdem wir zurück zum master Branch gewechselt haben, sehen wir, dass die Änderungen des new_feature Branches verschwunden sind. Stattdessen befindet sich der Code wieder in seinem ursprünglichen Zustand. Wir können den Fehler also beheben, einen Commit ausführen und die App in der neuen Version veröffentlichen, um den Emailschreiber zu besänftigen.

Anschließend wechseln wir wieder in unseren Feature Branch und arbeiten dort weiter, wo wir unterbrochen wurden.

```
1 git checkout new_feature
```

10. Befindet sich der Feature Branch in einem Zustand, der veröffentlicht werden soll, muss er nur mit dem master Branch vereinigt werden. Dazu führen wir einen Merge durch.

```
1 git checkout master
2 git merge new_feature
```

Wurden im master Branch keine weiteren Commits hinzugefügt, können die Commits des Feature Branches einfach angehängt werden (*Fast-Forward*). Gehen die Branches jedoch auseinander, versucht Git, die Änderungen zusammenzuführen. Mögliche Konflikte müssen dabei wie im Skript beschrieben im Code behoben werden.

Der Feature Branch kann anschließend gelöscht werden:

```
1 git branch -d new_feature
```

11. Versionskontrolle ist nicht nur für größere Projekte essentiell und hilft bei der Entwicklung jeder iOS App, sondern ermöglicht auch die Zusammenarbeit mehrerer Entwickler an einem Projekt. Im Skript ist dieses Thema kurz beschrieben und auch auf den Service GitHub ^[4] verwiesen, der weltweit von Entwicklern verschiedenster Plattformen verwendet wird, um an Projekten zusammenzuarbeiten.

⁴<http://www.github.com>

Auf GitHub ist bspw. auch dieses Skript zu finden. Ihr könnt das Repository auf der Webseite einsehen ^[5] und herunterladen.

Navigiert dazu in der Konsole zu dem Verzeichnis eurer Projekte für diesen Kurs und führt einen `git clone` aus:

```
1 cd path/to/directory
2 git clone https://github.com/iOS-Dev-Kurs/Skript
```

Das Repository wird dabei in das angegebene Verzeichnis heruntergeladen. Im Unterverzeichnis `dist/` findet ihr die gesetzten Dokumente dieses Kurses in aktueller Version. Auch die Links auf der Vorlesungsseite verlinken auf diese Dateien.

Prinzipiell könnt ihr nun lokal an dem Repository weiterarbeiten und Commits durchführen, diese jedoch nicht hochladen. Zur Zusammenarbeit an Programmierprojekten werden entweder entsprechende Berechtigungen vergeben, oder das **Fork** System auf GitHub verwendet.

Wenn ich nun Änderungen am Remote Repository auf Github vornehme, also bspw. eine neue Version dieses Skript veröffentliche, können ihr diese mit nur einem Befehl laden und mit dem lokalen `master` Branch zusammenführen:

```
1 git pull
```

Übungsaufgaben

6. Chatter

[2 P.]

In dieser Aufgabe schreiben wir zusammen an einer App!

Hinweis: Aus naheliegenden Gründen funktioniert es insgesamt besser, wenn ihr diese Aufgabe nicht alle erst am Sonntagabend erledigt...

- i. Auf GitHub verwalte ich das Repository der *chatter* App ^[6]. Ihr könnt es zwar in dieser Form mit `git clone` herunterladen, jedoch keine Änderungen im Original-Repository auf dem Server veröffentlichen. Stattdessen benötigt ihr einen eigenen **Fork**, mit dem ihr arbeiten könnt. Erstellt dazu zunächst einen GitHub Account ^[7], wenn ihr noch keinen habt.

⁵<https://github.com/iOS-Dev-Kurs/Skript>

⁶<https://github.com/iOS-Dev-Kurs/chatter>

⁷<https://github.com/join>

- ii. Loggt euch mit eurem GitHub Account ein und befolgt die Anweisungen der GitHub Dokumentation ^[8], um einen Fork des *chatter* Repositories ^[9] zu erstellen. Dort ist insbesondere auch beschrieben, wie ihr das Repository anschließend lokal klonet und das Original-Repository als Remote *upstream* hinzufügt.
- iii. Ihr habt das Projekt nun lokal auf eurem Mac. Es ist sowohl mit eurem Fork des Repositories (*origin*) als auch mit meinem Original-Repository (*upstream*) verbunden. Ihr könnt jetzt jederzeit die Änderungen herunterladen, die ich oder andere Teilnehmer unseres Kurses am Original-Repository vornehmen, indem ihr von *upstream* pullt:

```
1 git pull --rebase upstream master # Ein Rebase ist hier angebrachter  
  ↪ als ein Merge, im Skript ist der Unterschied kurz beschrieben
```

Dies solltet ihr häufig tun, unter anderem immer bevor ihr beginnt, an dem Projekt zu arbeiten.

- iv. Falls ihr es noch nicht getan habt, solltet ihr euren Namen und eure GitHub Email der Git Konfiguration hinzufügen, sodass die Commits ordentlich eurem Account zugeordnet werden

```
1 git config --global user.name "__dein_name__"  
2 git config --global user.email __deine_github_email__
```

- v. Öffnet nun das Projekt in Xcode. Die *chatter* App ist in der im Repository enthaltenen *README.md* Datei beschrieben. Diese und die Kommentare im Code sollen euch bezüglich der App als Referenz dienen. Ihr könnt euch die Projektdateien anschauen und die App im Simulator oder auf euren Geräten ausführen und ausprobieren.
- vi. Ihr habt nun sicherlich erkannt, worum es in der App geht: Instanzen verschiedener Subklassen von *Chatter* chatten miteinander. Dabei überschreiben die Subklassen jeweils nur die Implementierung weniger Methoden, die in der *Chatter* Klasse dokumentiert sind.

Eure Aufgabe ist es nun, eine eigene Subklasse zu schreiben und sie mithilfe von Git den anderen zur Verfügung zu stellen! Ihr könnt bspw. versuchen, einen bekannten Charakter darzustellen, oder etwas völlig neues erschaffen. Lasst eurer Kreativität freien Lauf!

Erstellt dazu einfach eine neue Subklasse von *Chatter* mit dem Namen eures Charakters und platziert die *.swift*-Datei im Xcode Projekt Navigator unter `chatter > Model > Chatters`.

⁸<https://help.github.com/articles/fork-a-repo>

⁹<https://github.com/iOS-Dev-Kurs/chatter>

- vii. Überschreibt nun die relevanten Methoden wie in der *README.md* Datei beschrieben. Hier könnt ihr einfach zufällige Chatnachrichten generieren, oder auch komplexere Mechaniken einbauen, sodass eine etwas natürlichere Konversation zustande kommt.

In eurer eigenen Subklasse könnt ihr dabei beliebig Code schreiben und bspw. Attribute einführen, um den Zustand eures Charakters darzustellen, wenn ihr möchtet. Wenn es nötig ist, könnt ihr auch die *Message* Klasse leicht anpassen. Achtet dabei jedoch unbedingt darauf, dass der Code für die anderen ebenfalls noch funktionieren muss!

- viii. Sichert eure Änderungen regelmäßig in Commits, wenn der Code fehlerfrei kompiliert:

```
1 git add filename # Achtet bitte darauf, nur Änderungen eurer
  ↳ Subklasse und nur wenn nötig Änderungen in anderen Dateien zu
  ↳ committen. Die project.pbxproj Datei enthält Informationen zu
  ↳ den Projektdateien – da ihr neue Dateien hinzugefügt habt, müsst
  ↳ ihr diese auch committen.
2 git status # häufig den Status prüfen
3 git commit -m"describe your changes here"
```

- ix. Die neuen Commits entstehen zunächst nur lokal. Mit eurem Fork des Repositories auf GitHub könnt (und solltet) ihr diese jedoch jederzeit abgleichen. Bei der Gelegenheit bietet es sich an, wie zuvor beschrieben zunächst die neuesten Änderungen aus dem Original-Repository herunterzuladen:

```
1 git pull --rebase upstream master
```

Das Repository enthält dann sowohl den aktuellen Stand des Original-Repositories, als auch eure Änderungen. Dies könnt ihr auf euren Fork auf GitHub hochladen:

```
1 git push origin master
```

- x. Wenn ihr mit eurer neuen Chatter Subklasse zufrieden sein, schickt mir eine *Pull Request*. So werden eure Änderungen in das Original-Repository integriert und tauchen auch bei den anderen Teilnehmern auf, wenn diese das nächste mal einen *git pull* durchführen.

GitHub beschreibt das System der Pull Requests recht ausführlich in ihrer Dokumentation ^[10]. Wie dort beschrieben, müsst ihr den Prozess dazu mit Klick auf den *Compare & review* Button initiieren, noch einmal die Änderungen prüfen und anschließend absenden. Ich erhalte dann eine Benachrichtigung und muss dem Merge mit dem Original-Repository nur noch großzügig zustimmen.

¹⁰<https://help.github.com/articles/using-pull-requests>

Ich bin gespannt auf eure Implementierungen!

Kapitel 4

View Hierarchie

Die iOS App Entwicklung orientiert sich konsequent am *Model-View-Controller Konzept* der Programmierung. Es ist nicht nur in Apples Frameworks wie UIKit rigoros umgesetzt sondern stellt auch die Grundlage für die weitere Konzeption unserer Apps dar und wird auch in vielen anderen Bereichen der Softwareentwicklung verwendet. Das Konzept ist im Skript beschrieben und sollte bei Entscheidungen zur Architektur einer App stets als Referenz verwendet werden.

Relevante Kapitel im Skript: Das Model-View-Controller Konzept

Wir betrachten nun zunächst die *View* Komponente des Konzepts. Im Skript wird die **UIView** Klasse des UIKit Frameworks vorgestellt, die mit ihren Subklassen die Anzeige von Interfaceelementen auf dem Bildschirm übernimmt. Wir werden in diesem Kapitel die *View Hierarchie* kennenlernen, **UIView** Objekte erstellen und anzeigen, sowie mithilfe des *Auto Layout* Konzepts das User Interface dynamisch anpassen.

4.1 Handgeschriebene View Hierarchie



View

Relevante Kapitel im Skript: View Hierarchie

1. Ihr könnt das MVC-Konzept noch nicht im Schlaf rezitieren? Unbedingt erstmal im Skript nachlesen!
2. Zuvor haben wir für Xcode Projekte ein Template mit vorkonfiguriertem Target, Storyboard und View Controller verwendet. Für den allgemeinen Gebrauch sind solche Templates sehr sinnvoll, doch nun möchten wir die Architektur einer iOS App einmal genauer untersuchen. Verwendet dafür das `viewhierarchy_bare` Projekt, das auf der Vorlesungsseite zur Verfügung steht.
3. Das Projekt beinhaltet nur ein Target `viewhierarchy` und zwei Dateien: eine Konfigurationsdatei `Info.plist` und eine mit dem *Entry Point* Attribut `@UIApplicationMain` als *Application Delegate* markierte Klasse `AppDelegate`. Im Skript könnt ihr euch über den Startprozess einer iOS App informieren.

Implementieren wir die `application:didFinishLaunchingWithOptions:` Methode des Application Delegates, wird diese am Ende des Startvorgangs aufgerufen:

```

1  @UIApplicationMain
2  class AppDelegate: UIResponder, UIApplicationDelegate {
3      var window: UIWindow?
4
5      func application(application: UIApplication,
6          ↪ didFinishLaunchingWithOptions launchOptions: [NSObject:
7          ↪ AnyObject]?) -> Bool {
8          NSLog("Hello World!")
9          return true
10     }
11 }

```

4. Da wir in der Info.plist Datei kein Storyboard angeben, wird beim Start der App einfach ein leeres `UIWindow` erstellt und angezeigt. Das können wir auch selbst tun:

```

1  func application(application: UIApplication,
2      ↪ didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
3      ↪ -> Bool {
4      let window = UIWindow(frame: UIScreen.mainScreen().bounds)
5      window.backgroundColor = UIColor.whiteColor()
6      window.makeKeyAndVisible()
7      self.window = window
8      return true
9  }

```

5. Nun können wir die View Hierarchie des angezeigten `UIWindow` mit weiteren Objekten der Superklasse `UIView` füllen und diese somit auf dem Bildschirm anzeigen:

```

1  func application(application: UIApplication,
2      ↪ didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
3      ↪ -> Bool {
4      let window = UIWindow(frame: UIScreen.mainScreen().bounds)
5      window.backgroundColor = UIColor.whiteColor()
6      window.makeKeyAndVisible()
7      self.window = window
8
9      let label = UILabel(frame: CGRect(x: 0, y: 50, width:
10         ↪ window.frame.size.width, height: 50))
11      label.text = "Hello World!"
12      label.backgroundColor = UIColor.redColor()
13      window.addSubview(label)

```

```

11
12     let button = UIButton(frame: CGRect(x: window.frame.size.width / 2 -
    ↪ 40, y: 120, width: 80, height: 44))
13     button.backgroundColor = UIColor.blackColor()
14     button.setTitle("Click!", forState: .Normal)
15     button.addTarget(self, action: "clickButtonPressed:",
    ↪ forControlEvents: .TouchUpInside) // Diese Methode ist das
    ↪ Äquivalent einer IBAction Verbindung
16     window.addSubview(button)
17
18     return true
19 }
20
21 func clickButtonPressed(sender: UIButton) {
22     println("Click!")
23 }

```

Übungsaufgaben

7. Color Match

[2+1 P.]

Verwendet das viewhierarchy_bare Projekt als Ausgangspunkt um (ohne den Interface Builder zu verwenden) eine einfache App zu schreiben. Denkt euch etwas eigenes aus (das kann jedoch schnell komplex werden), oder implementiert das folgende einfache Spiel:

Auf dem Bildschirm werden Buttons angezeigt, die jeweils eine Farbe als Hintergrund und den Namen einer (ggfs. anderen!) Farbe als Titel tragen. Man muss nun versuchen, auf solche Buttons zu tippen, deren Farbe und Titel übereinstimmen.

Bonus [+1]: Verwendet dabei Git, um euer Projekt in Form von Commits zu sichern und stellt es zur Abgabe auf GitHub zur Verfügung.

Nach dem MVC-Konzept trennen wir die Implementierung von Modell, Präsentation und Steuerung:

Model Erstellt eine Swift Datei `ColorWord.swift` und implementiert ein Struct `ColorWord` mit zwei Attributen `color: UIColor` und (Überraschung!) `word: String`. Das Struct benötigt zusätzlich einen Initializer `init()`, der eine zufällige Kombination erstellt, sowie ein Computed Attribute `isCorrect: Bool`, das prüft, ob Farbe und Wort zusammenpassen.

View Implementiert eine Subklasse `ColorButton: UIButton` in einer weiteren Swift Datei. Diese benötigt nur eine Methode `configureForColorWord:`, die eine Instanz `colorWord: ColorWord` als Argument annimmt und die `backgroundColor` und `title` Attribute, die von `UIButton` geerbt werden, entsprechend setzt.

Controller Der Steuerungscode wird normalerweise in Subklassen von `UIViewController` geschrieben, doch hier verwenden wir einfach die `AppDelegate` Klasse.

Fügt der `AppDelegate` Klasse die Attribute `var score: Int`, `var rounds: Int`, `var timer: NSTimer?`, sowie `var scoreLabel: UILabel!` und `var colorButtons: [ColorButton]` hinzu.

Das Spiel kann nun bspw. so ablaufen:

- a) In `application:didFinishLaunchingWithOptions:` wird das window und das `scoreLabel` erstellt und konfiguriert.
- b) Dann wird (ggfs. mehrmals) eine Methode `addColorButton` aufgerufen, die einen solchen Button erstellt und dem Array `colorButtons` sowie der View Hierarchie hinzufügt. Überlegt euch, wie ihr den Button positioniert. Wenn der Button betätigt wird, soll `colorButtonPressed:` aufgerufen werden.
- c) Die Methode `colorButtonPressed(sender: ColorButton)` muss dann natürlich implementiert werden. Prüft bspw., ob die Kombination des betätigten Buttons korrekt ist, um entsprechend Punkte von `score` abzuziehen oder hinzuzufügen. Anschließend soll eine Methode `prepareNextRound` aufgerufen werden.
- d) `prepareNextRound` fügt zunächst gegebenenfalls weitere Buttons mit `addColorButton` hinzu, um den Schwierigkeitsgrad zu erhöhen. Dann soll jeder angezeigte Button mit einem neuen `ColorWord` konfiguriert werden und ein Timer gestartet werden:

```

1 func prepareNextRound() {
2     self.rounds++
3     // ...
4     if let timer = self.timer {
5         timer.invalidate()
6     }
7     self.timer = NSTimer.scheduledTimerWithTimeInterval(/* Zeit
8     ↪ in sec */, target: self, selector: "timerFired:",
9     ↪ userInfo: nil, repeats: false)
10 }

```

- e) Die Methode `timerFired(timer: NSTimer)` wird nun nach Ablauf des Timers aufgerufen und muss implementiert werden. Hier können bspw. Punkte von der `score` abgezogen werden, wenn ein Button die korrekte Kombination trägt. Anschließend soll wieder `prepareNextRound` aufgerufen werden.
- f) `prepareNextRound` könnt ihr auch am Ende der `application:didFinishLaunchingWithOptions:` Methode bereits aufrufen, um den ersten Timer zu starten.
- g) Das `scoreLabel` muss an entsprechenden Stellen aktualisiert werden, am besten mithilfe einer Methode `updateScoreLabel`.

Um das Spiel interessant zu gestalten, gibt es natürlich an vielen Stellen noch Erweiterungspotential!

4.2 Auto Layout





View

Eine View Hierarchie können wir offenbar ebenso im Code schreiben wie im Storyboard konfigurieren. Selbst für ein simples Interface wie im vorherigen Abschnitt implementiert ist jedoch viel Code notwendig, da jeder Parameter als Attribut gesetzt werden muss. Der Interface Builder bietet hier effiziente Möglichkeiten, Benutzeroberflächen ohne Code zu konfigurieren und trotzdem mit dem Code zu verknüpfen.

Eine große Stärke des Interface Builders zeigt sich auch bei der Implementierung von dynamischen Benutzeroberflächen. Um auf Änderungen der Darstellung, wie bspw. Orientierungswechsel von Portrait auf Landscape, zu reagieren, müssten wir extensiv Code schreiben und die Frames der Views unserer View Hierarchie berechnen.

iOS Apps verwenden das *Auto Layout* Konzept von UIKit. Anstatt manuell Frames zu berechnen, definieren wir Regeln, die das Layout unserer Views erfüllen soll. Dieses Konzept der *Constraints* ist im Skript detailliert beschrieben. Mit *Size Classes* kann das Interface dann für bestimmte Displaygrößen genauer angepasst werden.

Relevante Kapitel im Skript: Auto Layout

1. Betrachten wir die RGB App als Beispiel für eine der zuvor konfigurierten einfachen Benutzeroberflächen. Ihr könnt auch das Projekt einer anderen App mit vergleichsweise einfachem Interface öffnen. Rotieren wir den Simulator mit  oder  in die Landscape Orientierung, werden die Frames der einzelnen Views nicht verändert und die Benutzeroberfläche wird nicht wie gewünscht angezeigt (s. S. 34, Abb. 4.1).

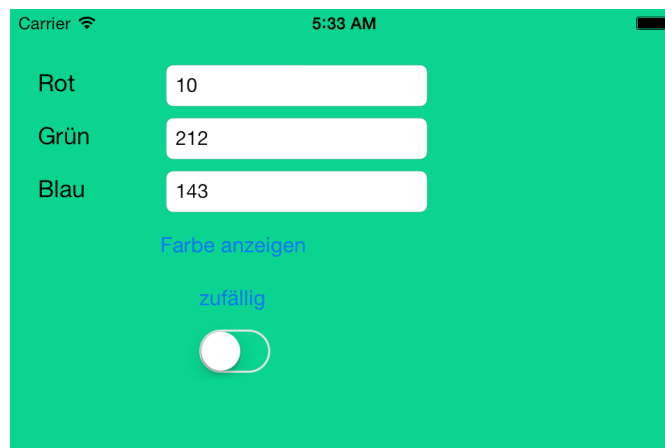


Abbildung 4.1: In der Landscape Orientierung bleiben die absoluten Frames einfach erhalten.

2. Um das Problem zu lösen, können wir nach dem Auto Layout Konzept nun Constraints definieren und damit `NSLayoutConstraint` Objekte der Superview hinzufügen. Zur Laufzeit positioniert die Superview ihre Subviews dann automatisch, sodass diese Constraints erfüllt sind. Die einfachste Möglichkeit zur Erstellung von Constraints bietet der Interface Builder.

Im Storyboard stellen wir zunächst sicher, dass Auto Layout für diese Interface Builder Datei aktiviert ist. Dazu muss die Option *Use Autolayout* im File Inspector markiert sein.

3. Nun können wir Constraints zwischen Interfaceelementen nach unseren Vorstellungen definieren. Dazu verwenden wir die Schaltflächen am rechten unteren Bildschirmrand oder ziehen Verbindungslinien zwischen Objekten bei gehaltener **⌘**-Taste. Im Skript sind die Möglichkeiten bei der Erstellung von Constraints beschrieben.
4. In einem eindeutigen Layout werden die Constraints blau markiert. Entsprechen nur die Frames nicht den Constraints, erscheinen diese in gelb und ihr könnt das *Resolve Auto Layout Issues* Menü am unteren rechten Bildschirmrand verwenden, um mit einem Klick auf *Update Frames* alle Views entsprechend ihrer Constraints zu positionieren. Bei einem eindeutigen Layout werden die Frames automatisch angepasst, wenn ihr die Konstanten der Constraints verändert. Sind Constraints rot gefärbt, gibt es Konflikte!

Fügt so lange passende Constraints hinzu, bis das Layout dadurch eindeutig beschrieben wird. **Überlegt euch dabei für jede Subview genau, welche Constraints ihr benötigt, um die vier Parameter *x*, *y*, *width* und *height* von *frame* eindeutig festzulegen.** Beachtet die *Intrinsic Content Size*!

5. Je nach Layout kann es nun sinnvoll sein, die Constraints für bestimmte *Size Classes* anzupassen. Sollen einige Interfaceelemente bspw. auf einem breiteren Display nebeneinander statt untereinander positioniert werden, wählen wir zunächst die *horizontal Regular*, *vertikal Any* Size Class mithilfe der Schaltfläche am unteren Rand des Editorbereichs.
6. Änderungen, die wir nun am Layout durchführen, betreffen nur die Anzeige in dieser Size Class. Wir können also die Interfaceelemente verschieben und Constraints verändern, bis uns das Layout für breite Displays gefällt.
7. Führt die App nun auf den iPhone und iPad Simulatoren aus und testet euer Layout bei verschiedenen Displaygrößen und -orientierungen.

Übungsaufgaben

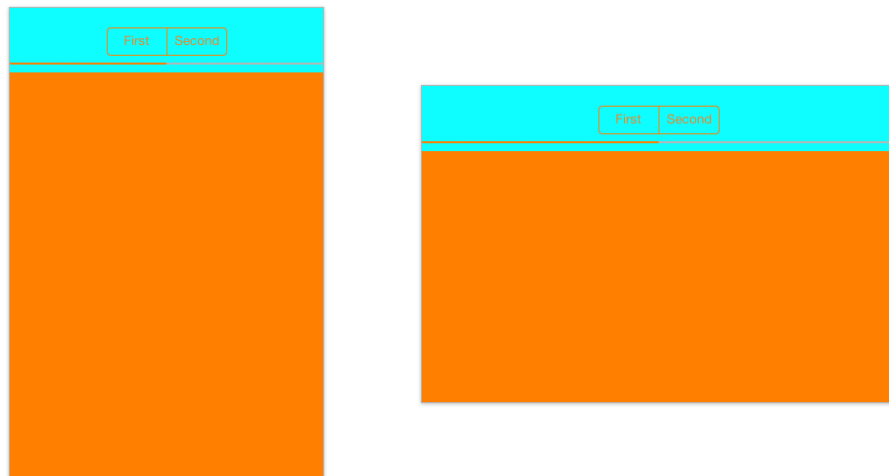
8. Auto Layout [3 P.]

Fügt eurer Counter, BMI oder RGB App oder einer vergleichbar einfachen App im Storyboard passende Constraints hinzu, sodass die Benutzeroberfläche sowohl in Portrait und Landscape Orientierung als auch bei verschiedenen Displaygrößen sinnvoll angezeigt wird. Dabei sollte das Layout eurer View Hierarchie eindeutig durch Constraints definiert sein. [1 P.]

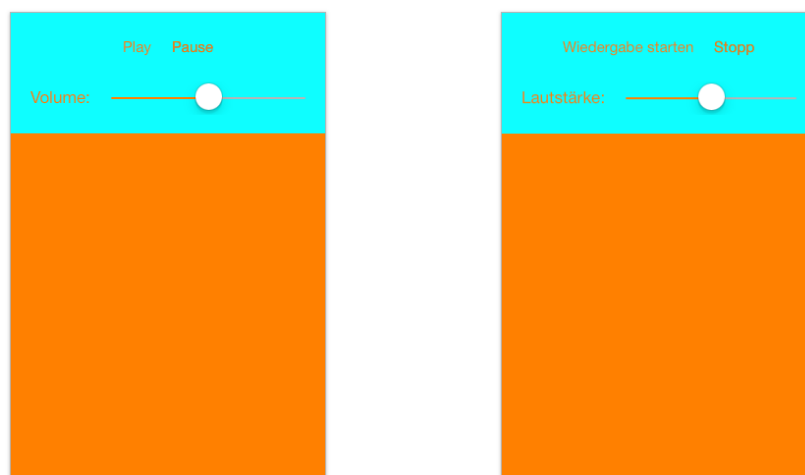
Löst dann die folgenden Layouts durch geschickte Definition von Constraints. [2 P.]

Hinweise:

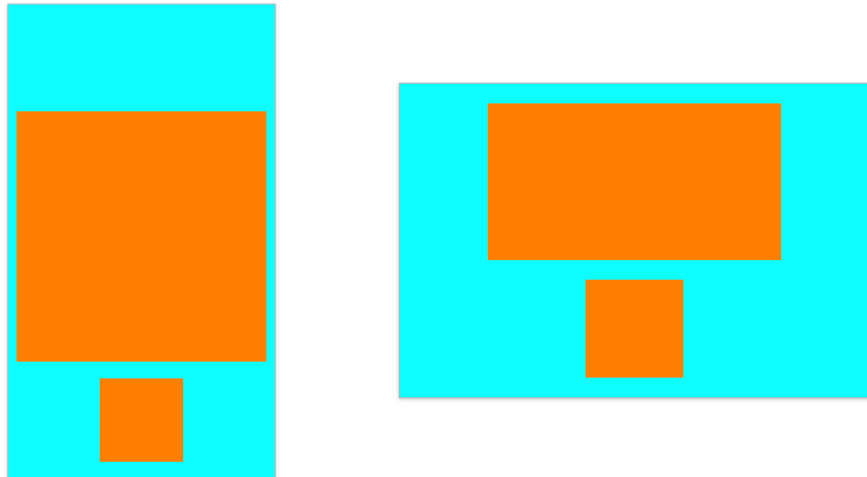
- Ihr könnt ein neues Projekt AutoLayout nach dem *Single View* Template erstellen und dem Storyboard einfach für jedes Problem ein *View Controller* Objekt aus der Object Library hinzufügen. Dessen View Hierarchie könnt ihr dann im Storyboard mit Views und Constraints konfigurieren. Mit den Schaltflächen im *Simulated Metrics* Abschnitt des *Attribute Inspectors*, könnt ihr bei ausgewähltem View Controller eine Gerätegröße und -orientierung simulieren.
 - In einigen Situationen kann die Verwendung von unsichtbaren Views als Platzhalter hilfreich sein. Dafür kann das Attribut `hidden` verwendet werden, das auch im Interface Builder verfügbar ist.
- a) Eine `UISegmentedControl` und eine `UIProgressView` sind am oberen Bildschirmrand positioniert, eine `UIView` füllt den verbliebenen Platz.



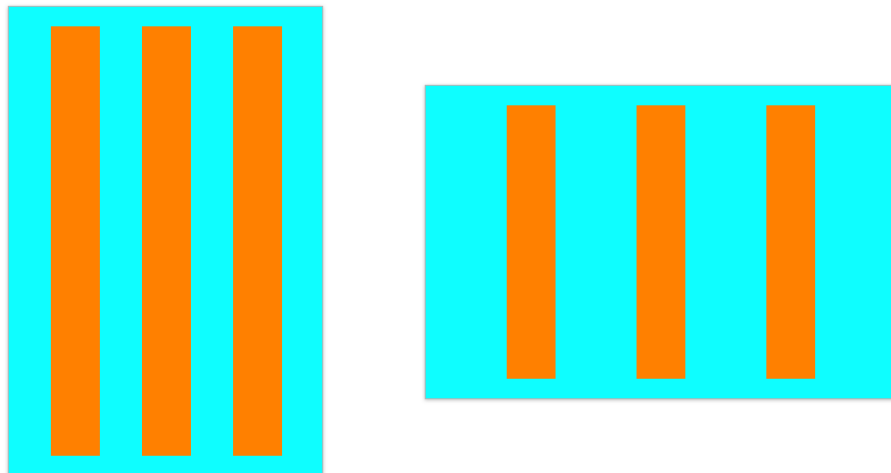
- b) Zwei Buttons im Abstand von 20pt sind am oberen Bildschirmrand **zusammen** horizontal zentriert. Ein `UISlider` mit zugehörigem Label und eine füllende View befinden sich darunter. Ändern wir den Text der Label, passt sich das Layout an.



- c) Zwei Views haben (Platzhalter-) Intrinsic Content Sizes von 300x300pt und 100x100pt. Die größere wird wenn möglich vertikal zentriert, hat jedoch immer einen Abstand von mindestens 20pt zur darunter befindlichen kleineren View. Beide sind horizontal zentriert. Die kleinere View ist außerdem immer mindestens 20pt vom unteren Bildschirmrand entfernt. Wird der verfügbare Platz kleiner, wird die größere View vor der kleineren gestaucht.



- d) Drei Views füllen jeweils den verfügbaren vertikalen Platz mit einem Abstand von 20pt zur Begrenzung. Horizontal sind sie gleichmäßig verteilt.



Kapitel 5

Cities



Controller

iOS Apps bestehen im Allgemeinen nicht nur aus einer, sondern aus einer Vielzahl von untereinander verbundenen Bildschirmansichten. Wir haben gelernt, dass jede Komponente der Benutzeroberfläche letztendlich von einem `UIView` Objekt in der View Hierarchie des übergeordneten `UIWindow` Objekts dargestellt wird.

Nun können wir die View Hierarchie einer komplexen App natürlich nicht wie im vorherigen Abschnitt zentral im App Delegate verwalten (s. S. 31, Übungsaufgabe 7). Um eine sinnvolle Struktur zu schaffen, implementieren wir stattdessen *View Controller*.

Diese sind der Controller-Komponente des Model-View-Controller Konzepts zugeordnet und im Skript ausführlich beschrieben.




Mit dieser App lernen wir das *View Controller Containment* Prinzip kennen, schreiben eigene `UIViewController` Subklassen und verwenden einige wichtige View Controller aus dem UIKit Framework.

Relevante Kapitel im Skript: View Controller Hierarchie

5.1 One City

Wir wollen zunächst einen Buttons mit dem Titel einer Stadt implementieren und Informationen über die entsprechende Stadt in einem separaten View Controller anzeigen, wenn der Benutzer auf den Button drückt.

1. Beginnen wir mit einem neuen Xcode Projekt nach dem *Single View* Template und mit dem Product Name *Cities*. Es wird damit automatisch eine Klasse `AppDelegate` und ein Storyboard hinzugefügt. Zusätzlich befindet sich bereits eine Subklasse View-Controller: `UIViewController` in unserem Projekt.
2. Die Klasse `ViewController` benennen wir zunächst in `CitiesViewController` um. Dazu muss sowohl die Swift-Datei in "`CitiesViewController.swift`" als auch der Klassenname in `CitiesViewController`: `UIViewController` umbenannt werden. Außerdem muss die Klasse im Storyboard angepasst werden (s.u.).

3. Im Storyboard finden wir eine Scene mit einem View Controller. Wie im Skript beschrieben, können wir die zu verwendende `UIViewController` Subklasse im Identity Inspector ( +  + ) einstellen. Gebt hier bei ausgewähltem View Controller unter `Custom Class` > `Class` den Namen "`CitiesViewController`" ein.

Zur Laufzeit wird also ein Objekt dieser Subklasse erzeugt und dem `UIWindow` Objekt als Root View Controller zugeordnet, da die Scene als *Initial Scene* gekennzeichnet ist. Damit wird die Content View dieses View Controllers zu Beginn der Ausführung der App der View Hierarchie des `UIWindow` Objekts hinzugefügt.

4. Ziehen wir nun `UIView` Objekte auf die Content View des View Controllers, werden diese zur View Hierarchie der Content View hinzugefügt. Der View Controller ist für die Verwaltung seiner Content View zuständig. Daher stellen wir Verbindungen in Form von `IBOutlet`s und `IBActions` her, um Zugriff auf die Objekte zu erhalten und um auf Benutzereingaben reagieren zu können.

Wir benötigen zunächst nur ein `UIButton` Objekt mit entsprechendem `IBOutlet`, das den Namen einer beliebigen Stadt anzeigen soll (s. S. 39, Abb. 5.1). Als Titel des Buttons könnt ihr auch zunächst generisch "`City name`" o.ä. wählen, da wir diesen im Code anpassen.

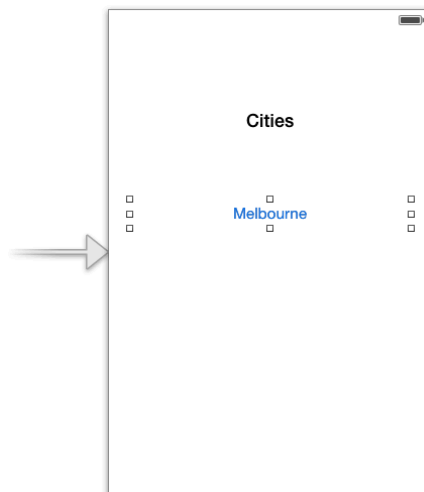


Abbildung 5.1: Ein Button soll bei Betätigung Informationen über die entsprechende Stadt anzeigen

5. Obwohl wir uns in dieser App hauptsächlich mit der Controller-Komponente beschäftigen, sollten wir eine einfache Datenstruktur implementieren, die zu unserer App passt. So können wir Daten einfacher verarbeiten und weitergeben. Erstellt also in einer neuen Swift Datei ein `struct City` und definiert die Attribute `let name: String` und `let image: UIImage?`.

```

1 // City.swift
2 import UIKit
3
```

```
4 struct City {  
5     let name: String  
6     let image: UIImage?  
7 }
```

6. Später wird der Cities View Controller eine Liste von Städten anzeigen, doch zunächst beschränken wir uns auf eine einzelne. Fügt CitiesViewController daher ein Attribut `var city: City?` hinzu:

```
1 // CitiesViewController.h  
2 import UIKit  
3  
4 class CitiesViewController: UIViewController {  
5     var city: City?  
6     @IBOutlet var cityButton: UIButton!  
7 }
```

7. Wir verwenden nun das App Delegate, um zu Beginn der Ausführung der App das Model, also die City Objekte, aufzusetzen und an die View Controller Hierarchie weiterzugeben. Implementiert daher folgende `application:didFinishLaunchingWithOptions:` Methode.

```
1 // AppDelegate.swift  
2 import UIKit  
3  
4 @UIApplicationMain  
5 class AppDelegate: UIResponder, UIApplicationDelegate {  
6     var window: UIWindow?  
7  
8     func application(application: UIApplication,  
9         ↪ didFinishLaunchingWithOptions launchOptions: [NSObject:  
10         ↪ AnyObject]?) -> Bool {  
11         let melbourne = City(name: "Melbourne", image: UIImage(named:  
12         ↪ "melbourne"))  
13         if let citiesViewController = window?.rootViewController as?  
14         ↪ CitiesViewController {  
15             citiesViewController.city = melbourne  
16         }  
17         return true  
18     }  
19 }
```

Es gibt natürlich auch Städte in Mitteleuropa, Panem und auf Naboo...

Hinweis: Der Initializer `init(named: String)` von `UIImage` lädt die Bilddatei mit dem angegebenen Dateinamen, sofern die Datei im Target referenziert ist. Um dem Target eine Bilddatei hinzuzufügen, könnt ihr sie einfach auf die Dateiliste des Project Navigators ziehen. Praktischer und ressourcenschonender ist jedoch die Verwendung von *Xcode Asset Catalogs* für die Bilddateien einer App. Eine solche mit Dateiendung `".xcassets"` ist bereits im Projekt enthalten. Diese Datei könnt ihr öffnen und auf den Bereich links im Editor per *Drag & Drop* einfach Bilddateien hinzufügen.

8. Im Cities View Controller überschreiben wir die `viewWillAppear:` Methode, um das Interface entsprechend des City Objekts zu konfigurieren.

```
1 // CitiesViewController.swift
2 import UIKit
3
4 class CitiesViewController: UIViewController {
5     var city: City?
6     @IBOutlet var cityButton: UIButton!
7
8     override func viewWillAppear(animated: Bool) {
9         cityButton.setTitle(city?.name, forState: .Normal)
10    }
11 }
```

Führt ihr die App an dieser Stelle mit einem *Build & Run* aus, wird der Button mit dem Namen der Stadt konfiguriert und angezeigt.

9. Um nun auf Knopfdruck einen Bildschirm zu präsentieren, der die Informationen der ausgewählten Stadt anzeigt, implementieren wir eine weitere Subklasse von `UIViewController`. Erstellt also eine neue Klasse `CityDetailViewController: UIViewController`. Dieser Klasse übergeben wir das ausgewählte City Objekt und überlassen ihr die Konfiguration ihrer Content View entsprechend den Attributen des Objekts. Definiert also wieder ein Attribut `var city: City?` im Header der `CityDetailViewController` Klasse.

```
1 // CityDetailViewController.h
2 import UIKit
3
4 class CityDetailViewController: UIViewController {
5     var city: City?
6 }
```

10. Nun verwenden wir das Storyboard, um die Benutzerführung zu konfigurieren. Zieht ein View Controller Objekt aus der Object Library auf das Storyboard und platziert es neben dem Cities View Controller. Wählt im Identity Inspector des hinzugefügten

View Controllers wie zuvor das Eingabefeld *Class* und gebt den Namen der neuen `UIViewController` Subklasse "`CityDetailViewController`" ein.

11. Platziert ein `UILabel` und ein `UIImageView` Objekt in der Content View des `CityDetailViewController` und verbindet sie mit IBOutlets im Code (s. S. 42, Abb. 5.2). Fügt außerdem einen "`Zurück`" Button hinzu.

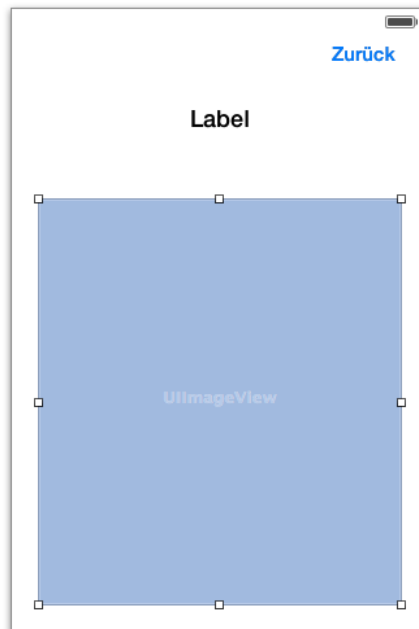


Abbildung 5.2: Der City Detail View Controller soll Information über die ausgewählte Stadt anzeigen

```

1 // CityDetailViewController.swift
2 import UIKit
3
4 class CityDetailViewController: UIViewController {
5     var city: City?
6
7     @IBOutlet weak var nameLabel: UILabel!
8     @IBOutlet weak var imageView: UIImageView!
9 }

```

12. Zur Präsentation des City Detail View Controllers können wir nun *Segues* verwenden. Diese repräsentieren, wie im Skript beschrieben, Beziehungen zwischen einzelnen Scenes im Storyboard. Segues können analog zu IBOutlets und IBActions erstellt werden, indem eine Verbindungslinie mit gedrückter `ctrl`-Taste gezogen wird. Wählt den Button im Cities View Controller aus und zieht eine Verbindung zum City Detail View Controller (s. S. 43, Abb. 5.3). Erstellt so eine *Modal Segue* zwischen Button

und View Controller (die Auswahlmöglichkeiten *show*, *show detail* und *present modally* haben in diesem Fall die gleiche Wirkung).

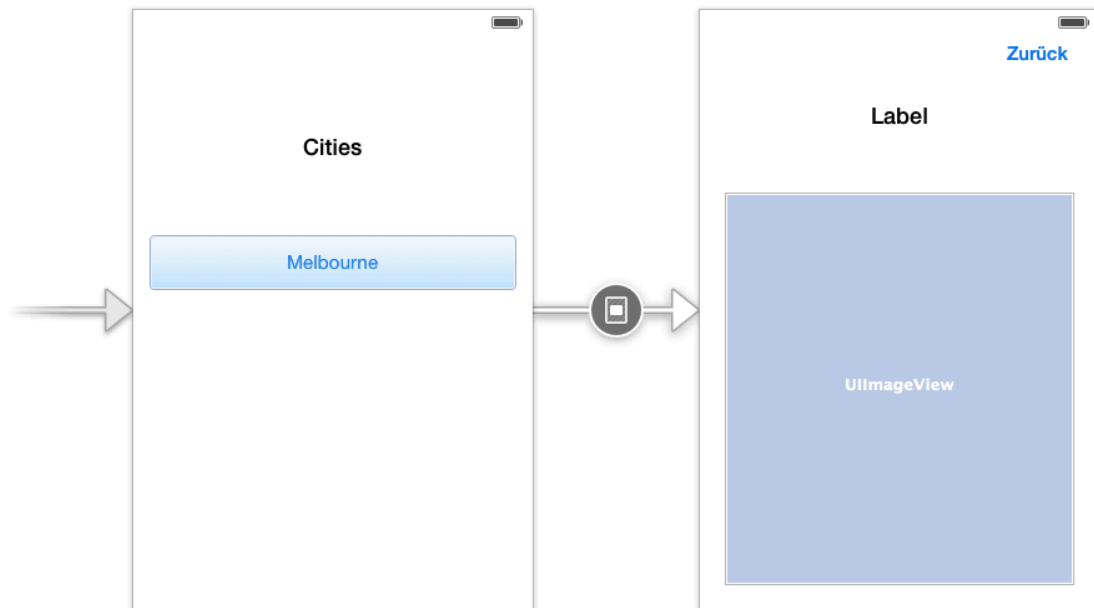


Abbildung 5.3: Eine Modal Segue konfiguriert die Präsentation des City Detail View Controllers bei Betätigung des Buttons

Im Attributes Inspector könnt ihr die Segue konfigurieren und bspw. zwischen verschiedenen Übergangsanimationen auswählen. Hier kann außerdem ein *Identifier* für die Segue vergeben werden. Setzt diesen auf `"showCityDetail"`.

13. Dem erstellten City Detail View Controller muss vor der Präsentation ein City Objekt übergeben werden, damit er dessen Attribute darstellen kann. Dazu implementieren wir die Instanzmethode `prepareForSegue(sender:)` in unserer `CitiesViewController` Klasse.

```

1 // CitiesViewController.swift
2 import UIKit
3
4 class CitiesViewController: UIViewController {
5     // ...
6     override func prepareForSegue(segue: UIStoryboardSegue, sender:
7     ↪ AnyObject?) {
8         if let identifier = segue.identifier {
9             switch identifier {
10                case "showCityDetail":
11                    if let cityDetailViewController =
12                        ↪ segue.destinationViewController as?
13                        ↪ CityDetailViewController {
14                        cityDetailViewController.city = self.city
15                    }
16            }
17        }
18    }
19 }

```

```

12         }
13         default:
14             break
15     }
16 }
17 }
18 }

```

Hier geben wir das City Objekt also einfach an den City Detail View Controller weiter.

14. Schließlich müssen wir den City Detail View Controller für die Darstellung der Attribute des City Objekts konfigurieren. Dies geschieht am besten in einer Implementierung der `viewWillAppear:` Instanzmethode in der `CityDetailViewController` Klasse.

```

1 // CityDetailViewController.swift
2 import UIKit
3
4 class CityDetailViewController: UIViewController {
5     var city: City?
6
7     @IBOutlet weak var nameLabel: UILabel!
8     @IBOutlet weak var imageView: UIImageView!
9
10    override func viewWillAppear(animated: Bool) {
11        self.nameLabel.text = city?.name
12        self.imageView.image = city?.image
13    }
14 }

```



Betätigt ihr nun nach einem *Build & Run* den Button, wird die Content View des City Detail View Controller angezeigt und mit dem entsprechenden City Objekt konfiguriert (s. S. 45, Abb. 5.4).

15. Nun fehlt nur noch die Implementierung des *Zurück* Buttons. Hier verwenden wir das Konzept der *Unwind Segue*, um zu einem View Controller in der View Controller Hierarchie zurückzukehren. Dazu fügen wir dem **präsentierenden** View Controller (nicht dem präsentierten) eine Methode `unwindToCities:` nach dem im Skript beschriebenen Muster hinzu, sodass sie im Interface Builder zur Verfügung steht.

```

1 // CitiesViewController.swift
2 import UIKit
3
4 class CitiesViewController: UIViewController {
5     // ...

```

Carrier  11:29 PM 
[Zurück](#)

Sydney



Abbildung 5.4: Wird der Button betätigt, zeigt der City Detail View Controller die Informationen zu der ausgewählten Stadt

```
6      @IBAction func unwindToCities(segue: UIStoryboardSegue) {  
7          // no implementation necessary  
8      }  
9  }
```

16. Im Storyboard können wir nun das `UIControlEvent.TouchUpInside` Event des *Zurück* Buttons mit der Unwind Segue verbinden. Zieht dazu bei gehaltener `[ctrl]`-Taste eine Verbindung vom Button zur *Exit* Schaltfläche des City Detail View Controllers (also des **präsentierten** View Controllers) (s. S. 46, Abb. 5.5). Da ein View Controller in der View Controller Hierarchie eine Unwind Segue `unwindToCities:` implementiert, könnt ihr diese hier auswählen.

Führt ihr die App nun aus und betätigt den *Zurück* Button, so wird die Unwind Segue zum Cities View Controller ausgeführt und damit der City Detail View Controller wieder ausgeblendet.

Übungsaufgaben

9. Cities

[2+1 P.]

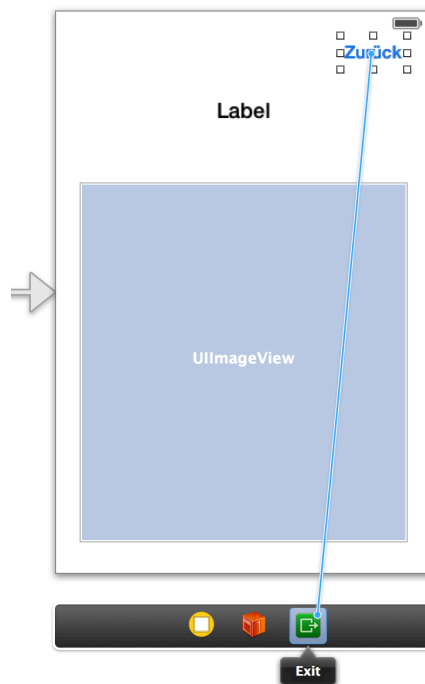


Abbildung 5.5: Die *Exit* Schaltfläche präsentiert alle Unwind Segues in der View Controller Hierarchie

Implementiert diesen ersten Teil der Cities App, indem ihr die Schritte oben nachvollzieht.

Bonus [+1]: Sichert das Projekt regelmäßig durch Commits in einem Git Repository und stellt es auf GitHub zur Verfügung.

5.2 One City Navigation

Eine solche Master-Detail View Controller Hierarchie, wie wir sie in Abschnitt 5.1 implementiert haben, ist ein so häufig verwendetes Konzept, dass UIKit dafür die Subklasse `UINavigationController`: `UIViewController` zur Verfügung stellt. Diese wird im Skript erläutert und eignet sich an dieser Stelle besser als Modal Segues.

1. Zieht einfach ein `UINavigationController` Objekt aus der Object Library auf euer Storyboard. Dabei wird zusätzlich zu der Navigation Controller Scene automatisch eine weitere View Controller Scene als Root View Controller des Navigation Controllers hinzugefügt. Löscht diesen zusätzlichen View Controller und wählt stattdessen den `CitiesViewController` als Root View Controller. Erstellt dafür eine *Relationship Segue* zwischen beiden Objekten, indem ihr wieder mit gedrückter `ctrl`-Taste eine Verbindung zieht. Markiert außerdem den Navigation Controller als Initial View Controller (s. S. 47, Abb. 5.6).

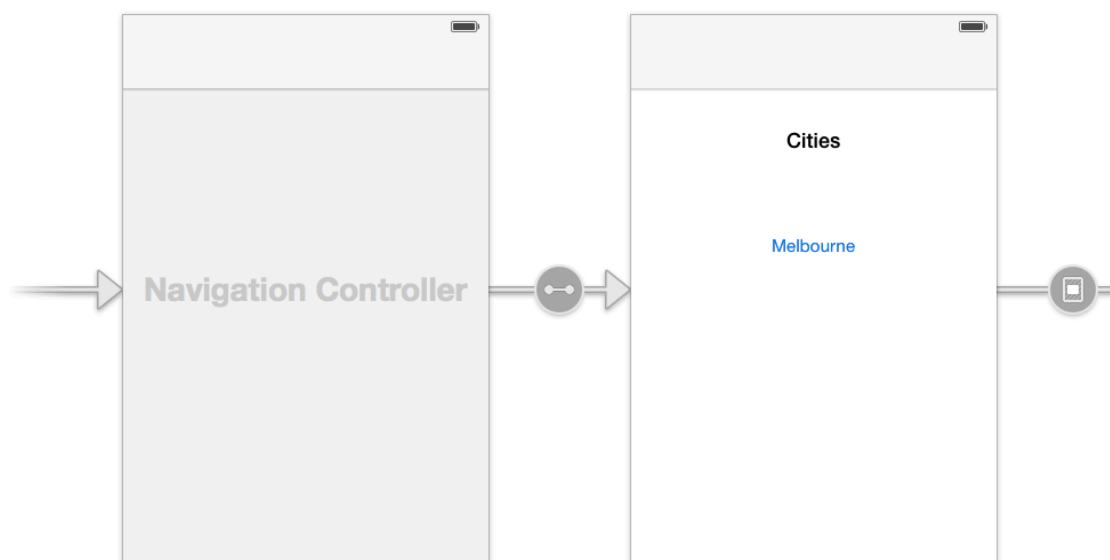


Abbildung 5.6: Eine Relationship Segue markiert den Root View Controller eines Navigation Controllers

2. Ihr könnt nun die Segue zwischen `CitiesViewController` und `CityDetailViewController` auswählen und im Attributes Inspector sicherstellen, dass der Typ *Show* ausgewählt ist. Wenn die Segue ausgelöst wird fängt dann der in der View Controller Hierarchie übergeordnete `UINavigationController` die Präsentation des `CityDetailViewController` ab.
3. Den *Zurück* Button könnt ihr nun entfernen, da `UINavigationController` einen eigenen Mechanismus implementiert und eine `UINavigationBar` am oberen Bildschirmrand anzeigt. Auch die Labels werden nicht mehr benötigt. Stattdessen können wir das `title: String` Attribut von `UIViewController` verwenden, dessen Wert als Titel in der Navigation Bar angezeigt wird (s. S. 48, Abb. 5.7).

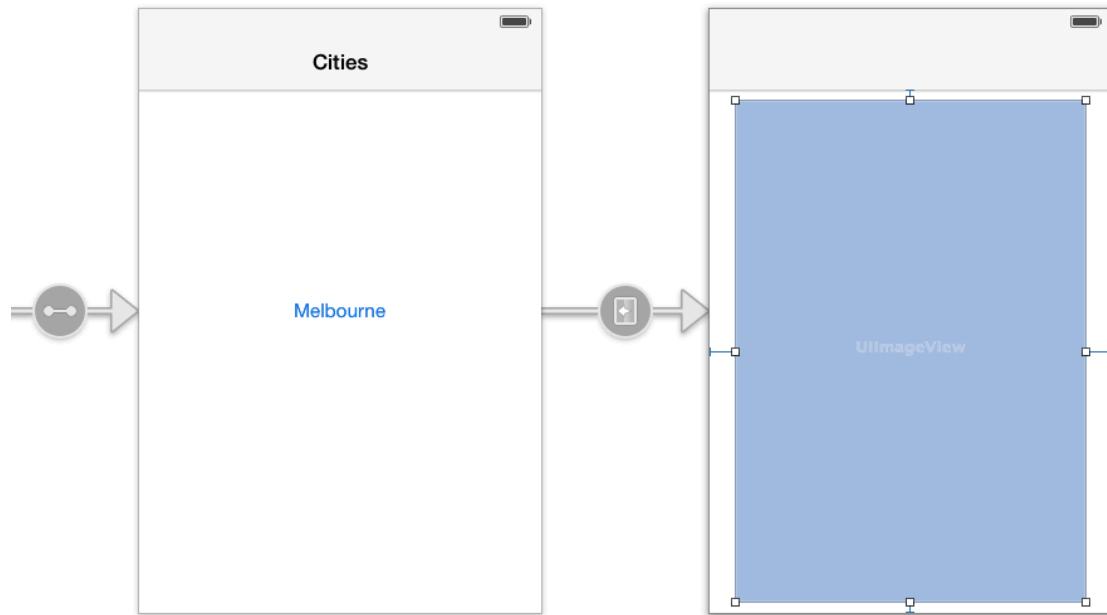


Abbildung 5.7: Navigation Controller zeigen eine Navigation Bar an

```

1 // CityDetailViewController.swift
2 import UIKit
3
4 class CityDetailViewController: UIViewController {
5     var city: City?
6
7     @IBOutlet weak var imageView: UIImageView!
8
9     override func viewWillAppear(animated: Bool) {
10         self.title = city?.name
11         self.imageView.image = city?.image
12     }
13 }

```

4. Da wir den Initial View Controller und damit den Root View Controller des Window Objekts verändert haben, müssen wir noch kurz die Implementierung des App Delegates anpassen, um das City Objekt korrekt weiterzugeben:

```

1 // AppDelegate.swift > application:didFinishLaunchingWithOptions:
2 let melbourne = City(name: "Melbourne", image: UIImage(named:
  ↳ "melbourne"))
3 if let citiesViewController = (window?.rootViewController as?
  ↳ UINavigationController)?.topViewController as? CitiesViewController {

```



```
4     citiesViewController.city = melbourne
5 }
```

5. Durch die Verwendung des Navigation Controllers erhält unsere App nun die Standardanimationen und -mechanismen aus `UIKit` (s. S. 49, Abb. 5.8), die sich bspw. durch die Verwendung von Subklassen und Implementierung von Delegate-Protokollen vielseitig anpassen lassen.

`UIKit` stellt noch weitere View Controller Container wie `UITabBarController` und `UISplitViewController` zur Verfügung, die nach einem ähnlichen Prinzip funktionieren, sowie eine vielseitige API zur Implementierung eigener View Controller Container.



Abbildung 5.8: Navigation Controller stellen Standardanimationen und -mechanismen zur Verfügung

5.3 More Cities

Wenn wir eine Liste von Objekten darstellen wollen, konfigurieren wir natürlich nicht explizit Views für jedes Objekt. Stattdessen verwenden wir Subklassen von `UITableViewController`: `UIViewController` und ihre Content Views der Klasse `UITableView`: `UIView`, die mit dem *Delegate Konzept* vielseitig einsetzbar sind.

Relevante Kapitel im Skript: Das Delegate Konzept, Table Views & Table View Controller

1. Wir möchten nun eine Liste von Städten anstatt einzelner Buttons anzeigen. Dafür ändern wir die Superklasse unseres `CitiesViewController` von `UIViewController` zu `UITableViewController`. Außerdem können wir die `IBOutlet` Referenz zu dem `city-Button` entfernen.
2. Ein Table View Controller hat ein Objekt der `UITableView` Klasse als Content View. Im Storyboard müssen wir daher die bisherige Content View mit einem Table View Objekt aus der Object Library ersetzen. Die Content View wird ersetzt, wenn wir die neue View einfach auf das View Controller Objekt in der Document Outline links ziehen.
- Hinweis:** Anstatt einen existierenden View Controller wie beschrieben in einen Table View Controller umzukonfigurieren, kann auch ein Table View Controller aus der Object Library gezogen werden. Dieser hat dann bereits eine `UITableView` als Content View.
3. Eine Table View kann, wie im Skript beschrieben, dynamischen oder statischen Inhalt darstellen. Im Attributes Inspector können wir den Modus *Dynamic Properties* auswählen. Damit die Table View nun Inhalt präsentieren kann, benötigt sie *Prototype Cells*, die den "Bauplan" für jede Zelle dieser Art definieren. Fügt dafür eine `UITableViewCell` aus der Object Library hinzu oder erhöht die entsprechende Zahl im Attributes Inspector um 1.
4. Wir können Prototype Cells nun entweder nach Belieben mit Subviews konfigurieren, oder im Attributes Inspector einen Standardstil auswählen. Wählt hier zunächst einfach *Basic* (s. S. 50, Abb. 5.9).

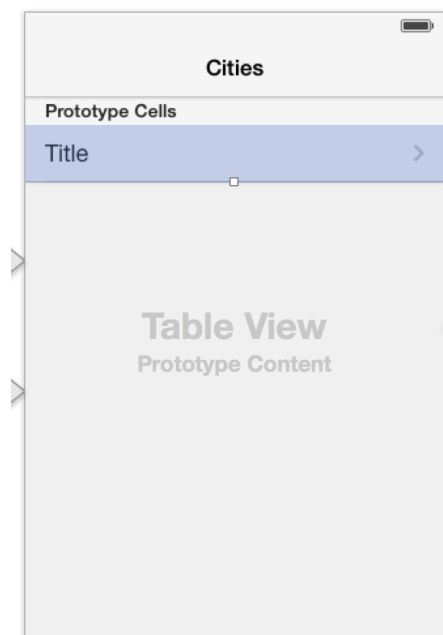


Abbildung 5.9: Prototype Cells dienen als Vorlage für die Zellen der Table View

5. Um Prototype Cells zu identifizieren, sollte ihnen ein *Reuse Identifier* im Attributes Inspector zugeordnet werden. Tippt hier "**cityCell**" ein. Wie im Skript erklärt kann die Table View damit Zellen, die gerade nicht angezeigt werden, an anderer Stelle wiederverwenden.
6. Da unsere Table View dynamischen Inhalt anzeigen soll, können wir diesen nicht im Storyboard konfigurieren. Stattdessen verwendet die Table View das *Delegate Konzept*, um Daten zu "erfragen", wenn sie sie benötigt. Dazu ruft das `UITableView` Objekt Methoden auf einem Delegate Objekt auf, die in einem *Protokoll* definiert sind. `UITableView` teilt diese Anfragen in die `UITableViewDataSource` und `UITableViewDelegate` Protokolle auf. Um den entsprechenden Attributen `datasource: UITableViewDataSource` und `delegate: UITableViewDelegate` Objekte zuzuweisen, sind diese als `IBOutlet`s markiert. Zieht also zwei Verbindungen von der Table View zum `CitiesViewController` und wählt diesen damit sowohl als Delegate als auch als DataSource.
7. Häufig repräsentiert eine Table View ein Array, wie hier eine Liste von Städten. Definiert also im öffentlichen Interface des `CitiesViewController` ein Attribut `cities: [City]?` anstatt des bisherigen `city: City?` Attributs.

Passt außerdem erneut die Implementierung des App Delegates an, um diesem Attribut nun eine Liste von Städten zuzuweisen:

```

1 // AppDelegate.swift
2 func application(application: UIApplication,
   ↳ didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
   ↳ -> Bool {
3     let melbourne = City(name: "Melbourne", image: UIImage(named:
   ↳ "melbourne"))
4     let sydney = City(name: "Sydney", image: UIImage(named: "sydney"))
5     if let citiesViewController = (window?.rootViewController as?
   ↳ UINavigationController)?.topViewController as?
   ↳ CitiesViewController {
6         citiesViewController.cities = [ melbourne, sydney ]
7     }
8     return true
9 }

```

8. Als Subklasse von `UITableViewController` erbt `CitiesViewController` leere Implementierungen der `UITableViewDataSource` und `UITableViewDelegate` Protokolle. Wir können diese nun überschreiben, um unsere Tabelle mit Daten zu füllen. Dabei sind zunächst die drei erforderlichen Methoden zur Darstellung einer dynamischen Table View zu implementieren:

```

1 // ViewController.swift
2 import UIKit
3

```

```

4  class CitiesViewController: UITableViewController {
5      var cities: [City]?
6
7      override func numberOfSectionsInTableView(tableView: UITableView) ->
        ↳ Int {
8          return 1
9      }
10     override func tableView(tableView: UITableView, numberOfRowsInSectionSection
        ↳ section: Int) -> Int {
11         return cities?.count ?? 0
12     }
13     override func tableView(tableView: UITableView, cellForRowAtIndexPath
        ↳ indexPath: NSIndexPath) -> UITableViewCell {
14         let city = cities![indexPath.row]
15         let cell =
            ↳ tableView.dequeueReusableCellWithIdentifier("cityCell",
            ↳ forIndexPath: indexPath) as! UITableViewCell
16         cell.textLabel?.text = city.name
17         cell.imageView?.image = city.image
18         return cell
19     }
20
21     override func prepareForSegue(segue: UIStoryboardSegue, sender:
        ↳ AnyObject?) {
22         if let indexPath = self.tableView.indexPathForSelectedRow() {
23             let city = cities![indexPath.row]
24             if let identifier = segue.identifier {
25                 switch identifier {
26                     case "showCityDetail":
27                         if let cityDetailViewController =
                            ↳ segue.destinationViewController as?
                            ↳ CityDetailViewController {
28                             cityDetailViewController.city = city
29                         }
30                     default:
31                         break
32                 }
33             }
34         }
35     }
36 }

```

-
9. Diese drei Methoden des UITableViewDataSource Protokolls stellen der Table View die erforderlichen Informationen zur Verfügung, um die Tabelle anzeigen zu können (s. S. 53, Abb. 5.10).
 10. Die Präsentation des Detail City View Controllers kann nun erneut mithilfe von Se-

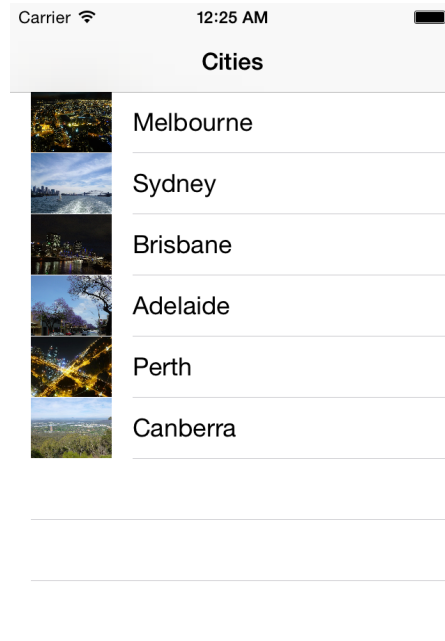


Abbildung 5.10: Eine Table View stellt Anfragen an ihr Datasource und Delegate Objekt, um Zellen dynamisch anzuzeigen

gues realisiert werden. Dazu können wir im Storyboard eine Show Segue von der Prototype Cell zum Detail City View Controller erstellen und ihr erneut den Identifier "**showCityDetail**" geben.

11. Da diese Segue nun von jeder Zelle ausgelöst wird, die nach Vorlage der Prototype Cell erstellt wurde, müssen wir in der `prepareForSegue:sender:` Methode wie im Codebeispiel oben zunächst den Index Path der betätigten Zelle herausfinden. Damit lässt sich anschließend das entsprechende Model Objekt erhalten und der City Detail View Controller konfigurieren.
12. Wird nun eine Stadt in der Liste ausgewählt, wird der City Detail View Controller entsprechend konfiguriert und angezeigt. Beachtet, dass wir an dessen Implementierung nichts geändert haben!

Übungsaufgaben

10. More Cities

[2 P.]

Implementiert die Cities App nun vollständig, indem ihr die Schritte oben nachvollzieht.