

UNIVERSITÄT HEIDELBERG

SOMMERSEMESTER 2014

Softwareentwicklung für iOS mit Objective-C und Xcode

APP KATALOG

NILS FISCHER

Aktualisiert am 23. Juni 2014
Begleitende Dokumente auf der Vorlesungsseite:
<http://ios-dev-kurs.github.io>

Inhaltsverzeichnis

1	Über dieses Dokument	3
2	Hello World	4
2.1	Das erste Xcode Projekt	4
2.2	@ "Hello World!"	5
2.3	@ "Hello World!" on Device	6
2.4	Grundlagen der Programmierung	7
2.5	Objektorientiertes @ "Hello World!"	9
2.6	Graphisches @ "Hello World!"	15
3	Versionskontrolle mit Git	21
4	View Hierarchie	27
4.1	Handgeschriebene View Hierarchie	27
4.2	Auto Layout	30
5	Cities	34
5.1	One City	34
5.2	One City Navigation	43
5.3	More Cities	45

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 Vorlesungsseite ^[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.

Beispiellösungen zu den Übungsaufgaben sind ebenfalls auf der Vorlesungsseite zu finden.

¹<http://ios-dev-kurs.github.io/>

Kapitel 2

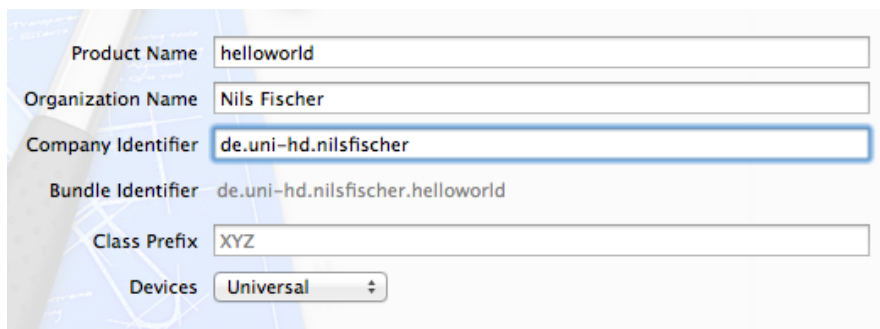
Hello World

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 eingeführt.

Relevante Kapitel im Skript: Xcode, Objective-C

2.1 Das erste Xcode Projekt

1. Mit `⌘ + ⌥ + N` rufen wir 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. 4, Abb. 2.1). Das führt zu der Bundle ID **de.uni-hd.deinname.helloworld**. Einen Class Prefix benötigen wir erstmal nicht. Speichert das Projekt in einem Verzeichnis eurer Wahl.



Product Name	helloworld
Organization Name	Nils Fischer
Company Identifier	de.uni-hd.nilsfischer
Bundle Identifier	de.uni-hd.nilsfischer.helloworld
Class Prefix	XYZ
Devices	Universal

Abbildung 2.1: Damit es keine Konflikte zwischen verschiedenen Apps gibt, gibt es Konventionen bei der Konfiguration

- 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. 5, Abb. 2.2).

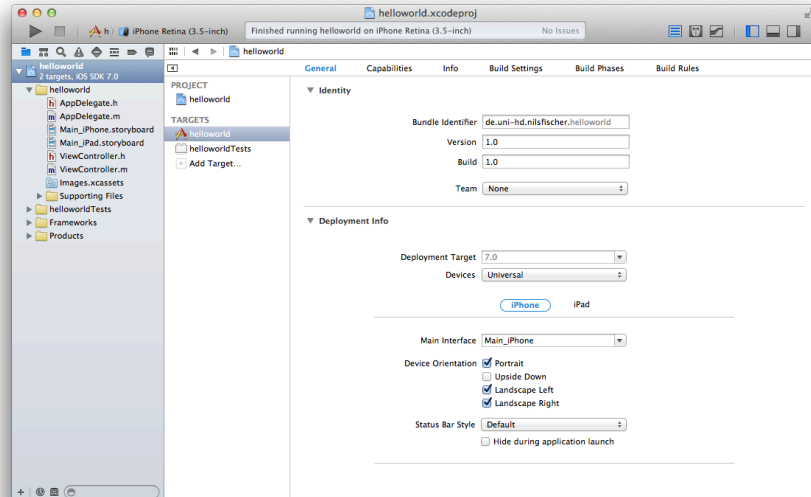


Abbildung 2.2: Wird das Projekt ausgewählt, sehen wir im Editor die Projekt- und Targetkonfiguration.

- 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.
- Links in der Toolbar sind die Steuerelemente des Compilers zu finden. Wählt das gerade erstellte Target und ein Zielsystem aus, bspw. den *iPhone Retina (3.5-inch)* Simulator, und klickt die *Build & Run* Schaltfläche. Das Target wird nun kompiliert und generiert ein *Product*, also unserer App, die im Simulator ausgeführt wird. Das kann bei der ersten Ausführung durchaus etwas dauern oder einen Fehler generieren. In Xcode kann mit $\text{⌘} + \text{⏏}$ die Ausführung gestoppt und mit $\text{⌘} + \text{R}$ (Tastenkürzel für *Build & Run*) dann neu gestartet werden.

2.2 @"Hello World!"

- Besonders spannend ist diese App natürlich noch nicht. Das ändern wir jetzt spektakulär, indem wir eine Ausgabe hinzufügen. Wählt die Datei *AppDelegate.m* im Project Navigator aus.
- Die Methode `application:didFinishLaunchingWithOptions:` wird zu Beginn der Ausführung der App aufgerufen. Zwischen den geschweiften Klammern ist bisher noch nicht viel zu finden:

```

1 - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
2     // Override point for customization after application launch.
3     return YES;
4 }

```

3. Ersetzt den Kommentar mit einem Befehl zur Ausgabe von Text in der Konsole:

```

1 - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
2     NSLog(@"Hello World!"); // Dieser Befehl gibt den Text Hello World! in
                             // der Konsole aus
3     return YES;
4 }

```

4. 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. 6, Abb. 2.3). 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.

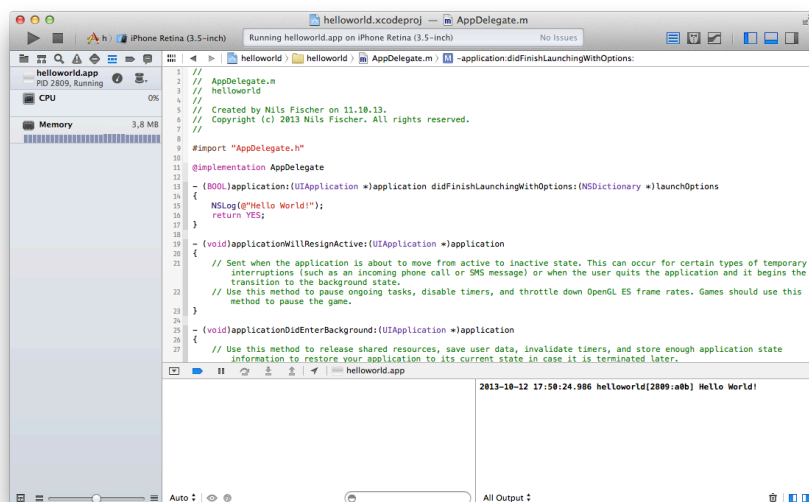


Abbildung 2.3: In der Konsole des Debug-Bereichs werden Ausgaben der laufenden App angezeigt

2.3 @"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.

- Habt ihr die Schritte befolgt und euren freigeschalteten Apple Developer Account in den Xcode-Accounteinstellungen hinzugefügt, öffnet ihr wieder die Project- und Targetkonfiguration im Project Navigator und wählt dort unser Developer Team (s. S. 7, Abb. 2.4) aus. Nun wird automatisch das richtige Provisioning Profile für die Bundle ID des Targets verwendet.

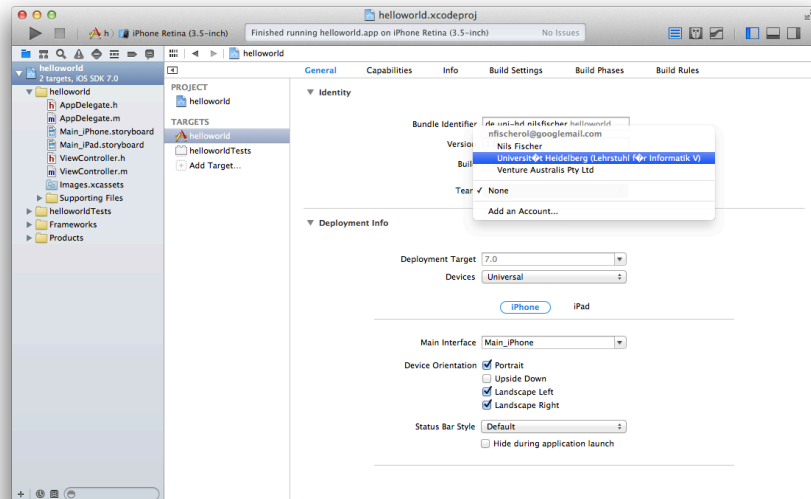


Abbildung 2.4: Mit der Wahl des zugehörigen Developer Teams in der Project- und Targetkonfiguration verwendet Xcode automatisch das passende Provisioning Profile

- 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.

2.4 Grundlagen der Programmierung

- Wir können nun beginnen, Objective-C Code zu schreiben. Öffnet dafür wieder die Datei *AppDelegate.m*.
- In der Methode `application:didFinishLaunchingWithOptions:`, die wir schon zuvor verwendet haben, können wir nun zunächst die Grundlagen der Programmierung wie im Skript beschrieben ausprobieren.

Übungsaufgaben

- Fibonacci

- 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

Schreibt einen Algorithmus, der alle Primzahlen $p_n < 1000$ in der Konsole ausgibt.

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

```
1  int a = 20%3 // a ist jetzt 2
```


2.5 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. Dazu brauchen wir zunächst eine neue Klasse `Person` und schreiben diese am besten in eine neue Datei. Mit dem Tastenkürzel `⌘ + N` rufen wir den *New File* Dialog auf.
2. Wählt hier `iOS > Cocoa Touch > Objective-C class` aus. Im nächsten Dialog können wir unsere neue Klasse konfigurieren. Wählt zunächst `NSObject` als Superklasse und gebt der Klasse den Namen `Person` (s. S. 9, Abb. 2.5).

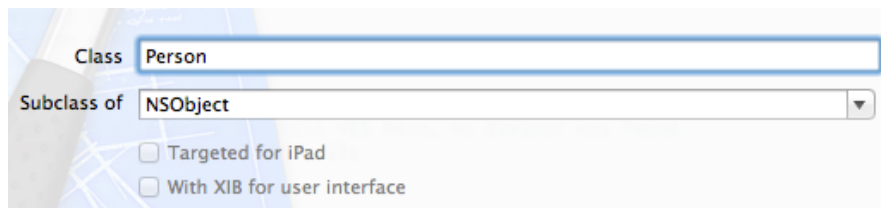


Abbildung 2.5: Der *New File* Dialog hilft bei der Konfiguration einer neuen Klasse

3. Stellt sicher, dass das Target *helloworld* im darauffolgenden Speicherdialog ausgewählt ist und speichert die Klasse im Projektverzeichnis.
4. Im Project Navigator sind nun zwei neue Dateien erschienen: Die Main- und die Header-Datei der neuen Klasse. Klickt auf die Main-Datei `Person.m`, um sie im Editor zu öffnen. Wenn ihr in der Toolbar nun anstatt des Standard- den Assistant-Editor auswählt, erscheint die Header-Datei `Person.h` automatisch auf der rechten Seite des Editors. Andernfalls klickt ihr auf die Jump bar des Assistant-Editors und wählt *Counterparts* aus, sodass die Header-Datei angezeigt wird (s. S. 10, Abb. 2.6).
5. Die neue Klasse soll Personen repräsentieren, die jeweils einen Namen besitzen. Die Header-Datei rechts im Assistant enthält das Interface der Klasse, also deren öffentliche Beschreibung. Hier definieren wir, dass jedes Objekt der Klasse 'Person' eine Variable `name` des Typs `NSString` haben soll. Außerdem soll die Klasse eine Methode mit dem Namen `sayHello` ohne Rückgabewert implementieren, die später den Gruß ausgeben soll:

```

1  @interface Person : NSObject // Das Interface der Klasse Person,
    Subklasse von NSObject, beginnt hier
2
3  @property (strong, nonatomic) NSString *name; // Jedes Objekt der Klasse
    besitzt dieses Attribut
4
5  - (void)sayHello; // Die Klasse implementiert diese Methode, die von
    jedem ihrer Objekte ausgeführt werden kann
6
7  @end

```

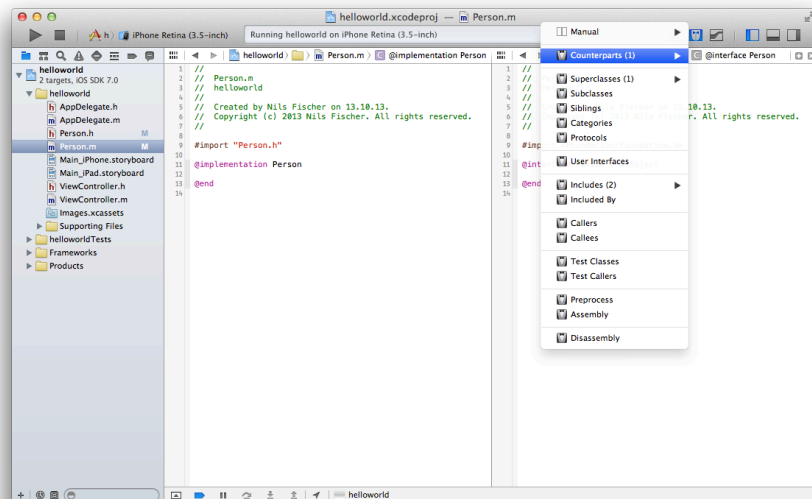


Abbildung 2.6: Der Assistant-Editor zeigt automatisch die Header-Datei zu einer geöffneten Main-Datei an, wenn die Option *Counterparts* gewählt wird

- Um zu bestimmen, was bei der Ausführung der Methode passiert, müssen wir sie noch implementieren. Dies geschieht in der Main-Datei *Person.m* links im Editor. Wir schreiben:

```

1 #import "Person.h"
2
3 @implementation Person
4
5 - (void)sayHello {
6     NSLog(@"Hello World! My name is %@.", self.name);
7 }
8
9 @end

```

Es wird also zusätzlich zu dem bekannten Gruß noch der Wert der Variable *name* in der Konsole ausgegeben. Dazu verwenden wir die dot-Syntax der Getter-Methode, die durch die Definition des Attributs *name* im Interface automatisch generiert wird.

- Unsere Klasse ist jetzt einsatzbereit und wir können Objekte nach ihrem Bauplan erstellen. Öffnen wir also wieder die Datei *AppDelegate.m*, in der wir auch zuvor die Grundlagen der Programmierung ausprobiert haben.
- Damit wir die Klasse verwenden können, müssen wir zunächst ihr Interface importieren. Fügt also den Befehl `#import "Person.h"` direkt über dem Beginn der Implementierung `@implementation AppDelegate` ein.
- Nun können wir Personen-Objekte erstellen. Wir verwenden wieder die Methode `application:didFinishLaunchingWithOptions` und schreiben:

```

1 #import "AppDelegate.h"

```

```

2
3  #import "Person.h" // Das Klasseninterface muss importiert werden, damit
                        die Klasse hier verfügbar ist
4
5  @implementation AppDelegate
6
7  - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
8
9      Person *aPerson = [[Person alloc] init]; // Ein neues Objekt der
                        Klasse Person wird erstellt
10     aPerson.name = @"Alice"; // Der Variable name dieses Objekts wird der
                        Wert @"Alice" zugewiesen.
11     [aPerson sayHello]; // Die Methode sayHello dieses Objekts wird
                        aufgerufen, in der auf die Variable zugegriffen wird
12
13     // Es können weitere, unabhängige Objekte nach dem gleichen Bauplan
                        der Klasse erstellt werden
14     Person *anotherPerson = [[Person alloc] init];
15     anotherPerson.name = @"Bob";
16     [anotherPerson sayHello];
17
18     return YES;
19 }
20
21 @end

```

10. Mit einem *Build & Run* führen wir die App aus und werden in der Konsole von Alice und Bob freundlich begrüßt:

```

1 Hello World! My name is Alice.
2 Hello World! My name is Bob.

```

11. Natürlich können wir unsere Klasse nun noch erweitern und Objekte miteinander interagieren lassen. Fügen wir also dem Interface der Klasse Person noch eine weitere Methode `sayHelloTo:` hinzu und implementieren sie:

```

1 // in der Header-Datei
2
3 @interface Person : NSObject
4
5 @property (strong, nonatomic) NSString *name;
6
7 - (void)sayHello;
8 - (void)sayHelloTo:(Person *)otherPerson;
9
10 @end
11
12 // in der Main-Datei
13
14 #import "Person.h"
15
16 @implementation Person
17
18 - (void)sayHello {
19     NSLog(@"Hello World! My name is %@.", self.name);

```

```

20 }
21
22 - (void)sayHelloTo:(Person *)otherPerson {
23     NSLog(@"Hi %@! My name is %@.", otherPerson.name, self.name);
24 }
25
26 @end

```

Die neue Methode nimmt ein Argument in Form eines anderen Objekts der Klasse Person an und gibt dessen Wert der Variable name zusätzlich in der Konsole aus.

12. In der application:didFinishLaunchingWithOptions-Methode fügen wir nun einen Aufruf dieser Methode hinzu:

```

1  Person *aPerson = [[Person alloc] init];
2  aPerson.name = @"Alice";
3  [aPerson sayHello];
4
5  Person *anotherPerson = [[Person alloc] init];
6  anotherPerson.name = @"Bob";
7  [anotherPerson sayHello];
8
9  [anotherPerson sayHelloTo:aPerson]; // Die Methode sayHelloTo: wird vom
    Objekt anotherPerson aufgerufen und es wird ihr das Objekt aPerson
    als Argument übergeben

```

Ausgabe:

```

1  Hello World! My name is Alice.
2  Hello World! My name is Bob.
3  Hi Alice! My name is Bob.

```

13. Abgesehen von den primitiven Datentypen, die wir bereits kennengelernt haben, sind viele Grundelemente der Programmierung in Objective-C Objekte. Im Skript werden einige wichtige beschrieben. Dazu gehört das (statische) `NSArray` und sein (veränderbares) Pendant `NSMutableArray`. Mit Arrays können wir Listen von Objekten erstellen:

```

1  NSArray *persons = @[aPerson, anotherPerson]; // Erstellt ein Objekt der
    Klasse NSArray mit den gegebenen Person-Objekten
2
3  for (Person *person in persons) { // Die Objekte im Array persons werden
    durchgegangen (enumerated)
4      [person sayHello];
5  }

```

oder:

```

1  NSArray *names = @[@"Alice", @"Bob", @"Cindy", @"Bruce", @"Chris", @"Bill",
    @"Susan"];
2
3  NSMutableArray *persons = [[NSMutableArray alloc] init]; // Ein verä
    nderbares Array wird erstellt
4

```

```

5  for (int i=0; i<[names count]; i++) { // Die Schleife wird für jeden
    Index des Arrays names ausgeführt
6    Person *newPerson = [[Person alloc] init];
7    newPerson.name = [names objectAtIndex:i]; // Einem neuen Person-Objekt
    wird das NSString-Objekt am aktuellen Index als Name
    zugewiesen
8
9    [persons addObject:newPerson]; // Das Person-Objekt wird der Liste
    persons hinzugefügt
10
11    [newPerson sayHello];
12 }

```

Übungsaufgaben

3. Scientists

- Erstellt eine weitere Klasse Scientist als Subklasse von Person.
- Wissenschaftler können rechnen, fügt dieser Klasse also eine Methode sayPrimeNumbersUpTo: hinzu, die ein Argument des Datentyps `int` annimmt und alle Primzahlen bis zu dieser Zahl in der Konsole ausgibt. Verwendet dazu den Algorithmus aus der vorherigen Übungsaufgabe (s. S. 8, Übungsaufgabe 2).
- 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.
- Nach dem Prinzip der **Polymorphie** soll ein Wissenschaftler einen anderen Gruß ausgeben als eine normale Person. Informiert euch über Polymorphie im Skript und überschreibt in der Scientist-Klasse die Methode sayHello, sodass zusätzlich I know all prime numbers! ausgegeben wird.

4. Emails

- Erweitert die Klasse Person zunächst um ein Freundschaftssystem

Jede Person besitzt ein (privates) Attribut `NSMutableArray*friends`, das eine Liste ihrer Freunde darstellt. Das Aufrufen einer Instanzmethode `makeFriendsWith:` fügt eine Person dieser Liste hinzu. Freundschaften werden immer in beide Richtungen geschlossen, also sollte die Methode `makeFriendsWith:` dieselbe Methode der anderen Person aufrufen.

Hinweis: Um hier Endlosschleifen zu verhindern kann die Instanzmethode `containsObject:` von `NSArray` hilfreich sein, die testet, ob ein Objekt bereits in der Liste enthalten ist. Beachtet außerdem, dass einer Liste erst erstellt werden muss, bevor ihr Objekte hinzugefügt werden können:

```

1  if (self.friends==nil) { // Testet, ob self.friends bereits ein
    Objekt hält
2    self.friends = [[NSMutableArray alloc] init]; // Weist dem
    Attribut self.friends eine neu erstellte Liste zu
3  }

```



- b) Erstellt eine neue Klasse `Email : NSObject`. Wir simulieren nun das Senden und Weiterleiten von Emails. Die neue Klasse `Email` benötigt nur eine Instanzmethode `sendTo:`, die eine Liste von Personen `NSArray*recipients` als Argument annimmt. Die Implementierung dieser Methode ruft `receiveEmail:` auf jedem Objekt der Liste auf.
- c) Erweitert die Klasse `Person` um die Instanzmethoden `sendEmail` und `receiveEmail` :.
- `sendEmail` sendet eine neue Email an die Liste der Freunde der Person. `receiveEmail` : akzeptiert ein Argument `Email *email` und leitet die Email an alle Freunde weiter.

Hinweis: Damit die Klassen `Email` und `Person` in der jeweils anderen Klasse verfügbar sind, müssen die Header gegenseitig importiert werden. Verwendet das Prinzip der **Forward Declaration**, damit dies nicht zu einer Endlosschleife führt.

- d) Verwendet die bekannte Methode `application:didFinishLaunchingWithOptions` :, um die Simulation zu starten. Erstellt eine Person `Person *me` mit eurem eigenen Namen und eine Liste `NSMutableArray*persons` mit weiteren Personen, beispielsweise mit den zuvor im Beispiel verwendeten Namen.

Stellt eine Freundschaftsverbindung zwischen `me` und jeder Person aus `persons` her, sowie zwischen solchen Personen mit gleichem Anfangsbuchstaben.

Hinweis: Die Instanzmethode `characterAtIndex:` von `NSString` gibt den entsprechenden Buchstaben als Datentyp `char` zurück und kann einfach mit dem Operator `==` mit einem anderen verglichen werden.

Fügt in den verschiedenen Methoden Konsolenausgaben hinzu, damit ihr den Verlauf der Simulation nachvollziehen könnt. Ein Aufruf `[me sendEmail]` soll nun die Simulation starten. Nach dem *Build & Run* könnte das Tastenkürzel  +  zum Stoppen der Ausführung sinnvoll sein...



Hinweis: Wenn die Simulation in einer Endlosschleife läuft, steht nach einer Weile nicht mehr genug Speicherplatz zur Verfügung, um den nächsten Methodenaufruf auszuführen ^[1]. Ihr erhaltet dann den berüchtigten `EXC_BAD_ACCESS` Fehler und die Ausführung der App bricht ab. In anderen Situationen, in denen keine Endlosschleife ausgeführt wird, ist dieser Fehler ein Hinweis auf fehlerhaftes Speichermanagement. Dazu gehört bspw. der Zugriff auf ein Objekt, das bereits (möglicherweise aufgrund der Kennzeichnung des Attributs als `weak`) aus dem Speicher entfernt wurde.

- e) **Extra:** Überlegt euch eine Erweiterung, sodass Emails sinnvoll als Spam erkannt und verworfen werden und nicht endlos weitergeleitet werden.

¹http://en.wikipedia.org/wiki/Stack_overflow

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.

1. Zur Gestaltung der Benutzeroberfläche oder User Interface (UI) verwenden wir den in Xcode integrierten Interface Builder (IB). Wir haben bei der Projekterstellung dieser App das *Single View*-Template ausgewählt und konfiguriert, dass sowohl iPhone als auch iPad unterstützt werden soll (Universal). Daher enthält das Projekt bereits ein Storyboard für beide Gerättypen. Wählt im Project Navigator die Datei *Main_iPhone.storyboard* aus.
2. Der Editor-Bereich zeigt nun den Interface Builder an. 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. 2.7).

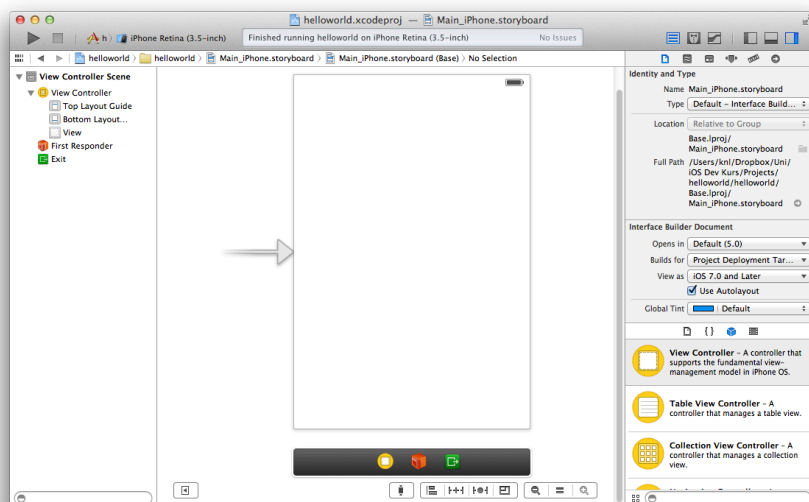


Abbildung 2.7: Für den Interface Builder verwenden wir eine angepasste Fensterkonfiguration mit dem Inspektor anstatt des Navigators

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 diesen Tab aus, wenn er noch nicht angezeigt wird.
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!`.

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, vom Typ **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. 2.8). Mit dem Attributes Inspector könnt ihr dem Button nun den Titel *Say Hello!* geben und für das Text Field einen Placeholder Name einstellen.

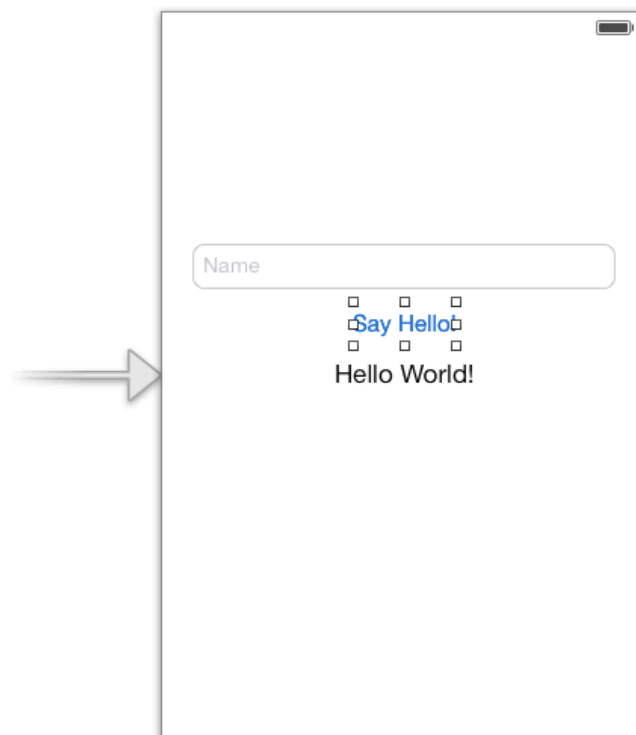


Abbildung 2.8: Mit einem Text Field, einem Button und einem Label erstellen wir ein simples UI

8. Nun müssen wir zur Laufzeit der App auf die erstellten Objekte zugreifen und auf Benutzereingaben reagieren können. Dazu verwenden wir sog. **IBOutlet**s 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 Main-Datei des übergeordneten View Controllers eingeblendet.
9. Da auf die Interfaceelemente nicht von außerhalb der Klasse zugegriffen werden muss, können wir das private Interface oben in der Main-Datei verwenden. Schreibt dort:


```

1 @interface ViewController ()
2
3 @property (strong, nonatomic) IBOutlet UITextField *nameTextField;
4 @property (strong, nonatomic) IBOutlet UILabel *helloLabel;
5
6 - (IBAction)sayHelloButtonPressed:(id)sender;
7
8 @end

```

Wir definieren also mit `IBOutlet` gekennzeichnete Attribute für das Text Field und das Label, die zur Laufzeit der App unsere erstellten Interfaceelemente halten sollen. Auf den Button müssen wir nicht zugreifen sondern nur das Event abfangen, wenn ein Benutzer darauf drückt. Also benötigen wir nur eine mit `IBAction` gekennzeichnete Methode, die ausgeführt wird, wenn das Event eintritt.

10. Nun zieht mit gedrückter `ctrl`-Taste eine Linie von dem Text Field und dem Button im Interface Builder auf das jeweilige Attribut im Code. Die Codezeile wird dabei blau hinterlegt. Zieht außerdem genauso eine Line von dem Button auf die eben definierte Methode. Im Connection Inspector könnt ihr die `IBOutlet`s und `IBAction`s eines ausgewählten Objekts betrachten und wieder entfernen.
11. Wenn der Benutzer auf den Button drückt wird nun die Methode `sayHelloButtonPressed` : ausgeführt. Diese müssen wir jedoch erst implementieren:

```

1 #import "Person.h"
2
3 @implementation ViewController
4
5 - (IBAction)sayHelloButtonPressed:(id)sender {
6     Person *newPerson = [[Person alloc] init];
7     newPerson.name = self.nameTextField.text;
8     [newPerson sayHello];
9 }
10
11 @end

```

Die Klasse `UITextField` besitzt ein Attribut `text` des Typs `NSString`. Wir verwenden hier ihre Getter-Methode in der Dot-Syntax, um den Inhalt des Text Fields zu erhalten.

Nach einem *Build & Run* könnt ihr im Text Field einen Namen eintippen und werdet in der Konsole von einer virtuellen Person diesen Namens begrüßt. Entfernt vorher am besten den Code in der `application:didFinishLaunchingWithOptions:`, wenn ihr es noch nicht getan habt, damit diese Ausgaben nicht stören.

12. Damit der Gruß auf dem Bildschirm ausgegeben werden kann, benötigen wir eine Methode, die den Gruß nicht in der Konsole ausgibt sondern als Rückgabewert zurückgibt. Dazu wechseln wir wieder in die Konfiguration mit Project Navigator und ausgeblendetem Inspektor und wählen die `Person.m` Datei.
13. Fügt dem Interface der Klasse `Person` die Definition der neuen Methode `helloString` mit Rückgabotyp `NSString` hinzu:

```
1 - (NSString *)helloString;
```

Diese müssen wir in der Main-Datei implementieren:

```
1 #import "Person.h"
2
3 @implementation Person
4
5 - (NSString *)helloString {
6     return [NSString stringWithFormat:@"Hello World! My name is %@.", self
7         .name];
8 }
9
10 - (void)sayHello {
11     // Um nicht zwei nahezu gleiche Methoden implementiert zu haben, rufen
12     // wir hier stattdessen die neue Methode auf
13     NSLog([self helloString]);
14 }
15
16 @end
```

Die Klasse `NSString` besitzt eine Klassenmethode `stringWithFormat:`, die ein neues `NSString`-Objekt nach dem gleichen String-Formatierungs-Prinzip zurückgibt, das wir bereits aus `NSLog()` kennen.

14. Wählt nun im Project Navigator die Datei *ViewController.m*. Hier haben wir zuvor die Methode `sayHelloButtonPressed:` implementiert. Nun setzen wir jedoch anstatt `sayHello` aufzurufen den angezeigten Text des Labels auf den Rückgabewert der neuen `helloString` Methode:

```
1 - (IBAction)sayHelloButtonPressed:(id)sender {
2     Person *newPerson = [[Person alloc] init];
3     newPerson.name = self.nameTextField.text;
4     self.helloLabel.text = [newPerson helloString];
5 }
```

15. Mit einem *Build & Run* erhalten wir unser erstes interaktives User Interface (s. S. 19, Abb. 2.9)!

Übungsaufgaben

5. Scientists 2

Überlegt euch, wie die Subklasse `Scientist` von `Person` angepasst werden muss, damit die Methoden `sayHello` und `helloString` die richtigen Ergebnisse liefern. Ändert die Klasse des Objekts, das bei Drücken des Buttons erzeugt wird, zu `Scientist`.

Hinweis: Die Instanzmethode `stringByAppendingString:` von Objekten der Klasse `NSString` gibt ein neues `NSString`-Objekt zurück, das aus dem Text des Empfängerobjekts mit angefügtem Text des Arguments besteht:

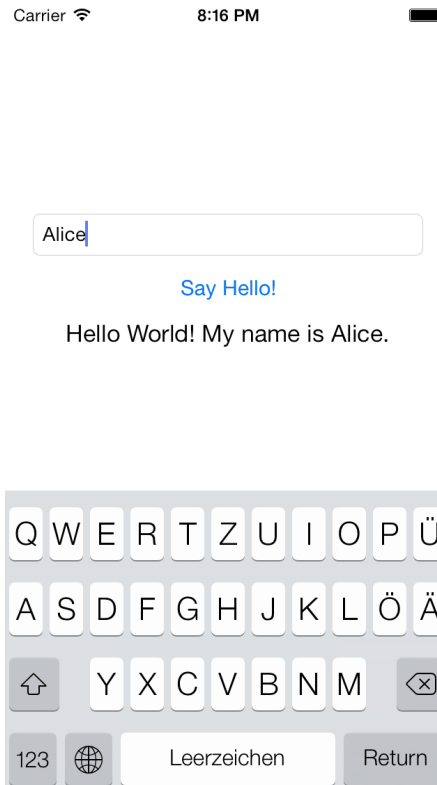


Abbildung 2.9: Drücken wir auf den Button, grüßt uns eine virtuelle Person mit dem angegebenen Namen. Sehr praktisch!

```
1 NSString *combinedString = [@"first" stringByAppendingString:@"second"];
2 // Wert von combinedString: firstsecond
```

Das gleiche Resultat lässt sich aber auch mit `stringWithFormat:` erzielen.

6. Simple UI

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

Lasst eurer Kreativität freien Lauf **oder** implementiert eines der folgenden Beispiele.

Counter Auf dem Bildschirm ist ein Label zu sehen, das den Wert einer Property `int count` anzeigt, wenn eine Methode `updateLabel` aufgerufen wird. Buttons mit den Titeln `+1`, `-1` und `Reset` ändern den Wert dieser Property entsprechend und rufen die `updateLabel`-Methode auf.

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

²<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.deiname.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!
- Es ist ausreichend, das User Interface entweder für iPhone oder iPad zu konfigurieren.
- `NSString` besitzt Instanzmethoden wie `floatValue` zur Umwandlung von Text in Zahlenwerte.
- Die Klassenmethode `colorWithRed:green:blue:alpha:` von `UIColor` nimmt Werte zwischen 0 und 1 an.
- Die Funktion `arc4random_uniform(n)` gibt eine Pseudozufallszahl x mit $0 \leq x < n$ aus.
- Wenn ein `UISwitch` betätigt wird, sendet dieser ein Event `UIControlEventValueChanged`, so wie ein `UIButton` das Event `UIControlEventTouchUpInside` sendet. Dieses Event kann genauso mit einer `IBAction` verbunden werden. Mit einem Klassenattribut `NSTimer*randomTimer` können wir dann die Methode für das zufällige Wechseln der Hintergrundfarbe implementieren:

```
1 - (IBAction)switchValueChanged:(UISwitch *)sender {
2     if (sender.isOn) {
3         self.randomTimer = [NSTimer scheduledTimerWithTimeInterval:0.1
4                               target:self selector:@selector(randomButtonPressed:)
5                               userInfo:nil repeats:YES];
6     } else {
7         [self.randomTimer invalidate];
8         self.randomTimer = nil;
9     }
10 }
```

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 können wir 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 `cd` 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.

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

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

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.m* und fügt ihrer Implementierung folgendes Codesegment hinzu:

```
1 - (void)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.m # oder git add --all
2 git commit -m "Changed background color"
```

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

³<http://git-scm.com/book/>

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 dass andere Branches wie der zuvor verwendete master Branch verändert werden. Implementiert bspw. wieder eine `viewDidLoad` Methode in der `ViewController.m` Datei und führt einen Commit durch:

```
1 - (void)viewDidLoad {
2     [super viewDidLoad];
3
4     UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"New Feature!"
5         " message:@"Is it a bug or a feature?" delegate:nil
6         cancelButtonTitle:@"Bug" otherButtonTitles:@"Feature", nil];
7     [alert show];
8 }
```

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

9. Erhalten wir nun plötzlich eine Email eines aufgebrachten Benutzers unserer App, der einen Bug gefunden hat, können wir problemlos zurück zum master Branch wechseln und den Bug 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 Bug 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.

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 `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/current/` 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

7. Chatter

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...

⁴<http://www.github.com>

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

- 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.
- 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. 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.
- v. Ihr habt nun sicherlich erkannt, worum es in der App geht: Instanzen verschiedener Subklassen von `CHTRChatter` chatten miteinander. Dabei überschreiben die Subklassen jeweils nur die Implementierung weniger Methoden, die in der `CHTRChatter` 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 `CHTRChatter` mit dem Namen eures Charakters und platziert die beiden zugehörigen Dateien im Xcode Projekt Navigator unter `chatter > Model > Custom Subclasses`.

- vi. 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.

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

⁷<https://github.com/join>

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

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

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 `CHTRMessage` Klasse leicht anpassen. Achtet dabei jedoch unbedingt darauf, dass der Code für die anderen ebenfalls noch funktionieren muss!

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

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

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, mü
   sst ihr diese auch committen.
2 git status # häufig den Status prüfen
3 git commit -m"describe your changes here"
```

- viii. Die neuen Commits sind zunächst nur lokal verfügbar. 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
```

- ix. Wenn ihr mit eurer neuen `CHTRChatter` 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.

Ich bin gespannt auf eure Implementierungen!

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

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 Foundation und **UIKit** rigoros umgesetzt sondern stellt auch die Grundlage für die weitere Konzeption unserer Apps dar und wird auch in vielen anderen Bereichen der Programmierung 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 kennt das MVC-Konzept noch nicht im Schlaf? Zeit, einen Blick ins Skript zu werfen!
2. Zuvor haben wir ein Storyboard verwendet, um eine einfache Benutzeroberfläche zu gestalten. Dabei haben wir die View Hierarchie unserer App mit dem Interface Builder konfiguriert. Natürlich können wir stattdessen oder ergänzend auch Code schreiben.
3. Erstellt ein neues Projekt mit dem Product Name *viewhierarchy*. Verwendet das *Empty Application* Template, sodass keine Interfaceelemente vorkonfiguriert werden.
4. Betrachtet nun wieder die `application:didFinishLaunchingWithOptions:` Methode des Application Delegates. Aus dem Skript (*iOS App Lifecycle*) wissen wir, dass diese Methode zu einem bestimmten Zeitpunkt im Startprozess der App aufgerufen wird.

5. Da wir mit dem *Empty Application* Template begonnen haben und kein Storyboard existiert, wird hier in einem Attribut `UIWindow*window` ein neues Objekt instanziiert und mit weißem Hintergrund angezeigt:

```

1 - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
2
3     self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen]
        bounds]];
4     self.window.backgroundColor = [UIColor whiteColor];
5     [self.window makeKeyAndVisible];
6
7     return YES;
8 }

```

6. Hier können wir nun die View Hierarchie mit weiteren Objekten der Superklasse `UIView` füllen und diese somit auf dem Bildschirm anzeigen:

```

1 - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
2
3     self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen]
        bounds]];
4     self.window.backgroundColor = [UIColor whiteColor];
5     [self.window makeKeyAndVisible];
6
7     UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(0, 50, self
        .window.frame.size.width, 50)];
8     label.text = @"Hello World!";
9     label.backgroundColor = [UIColor redColor];
10    [self.window addSubview:label];
11
12    return YES;
13 }

```

Übungsaufgaben

8. View Hierarchie

Implementiert eine der Apps **Counter**, **BMI** oder **RGB** aus der Übungsaufgabe des vorherigen Abschnitts (s. S. 19, Übungsaufgabe 6) oder eine vergleichsweise einfache App, ohne den Interface Builder zu verwenden.

Hinweise:

- Das Äquivalent zu einer IBOutlet Verbindung ist eine einfache Zuweisung eines `UIView` Objekts zu einer Property.
- Eine IBAction Verbindung hingegen ist durch die Instanzmethode `addTarget:action:forControlEvents:` von `UIControl` gegeben. `UIControl`: `UIResponder`: `UIView` ist bspw. die Superklasse von `UIButton`. Die Verwendung dieser Methode ist in der Dokumentation beschrieben und kann bspw. so aussehen:

```
1 [button addTarget:self action:@selector(buttonPressed:)
   forControlEvents:UIControlEventTouchUpInside];
```

Der Methode wird also die Methode als spezielles selector-Objekt übergeben, die bei dem angegebenen Event auf dem Target aufgerufen werden soll.

- Ein `UIButton` besitzt das Attribut `UILabel*titleLabel`, doch es gilt die Konvention, die Instanzmethode `setTitle:forState:` zu verwenden. Für verschiedene Modi wie `UIControlStateNormal` oder `UIControlStateSelected` können damit unterschiedliche Titel angegeben werden. Wird kein anderer Titel spezifiziert, wird der Titel von `UIControlStateNormal` verwendet.

4.2 Auto Layout



View

Eine View Hierarchie können wir offensichtlich 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 Objective-C. Anstatt manuell Frames zu berechnen, definieren wir Regeln, die das Layout unserer Views erfüllen soll. Dieses Konzept der **Constraints** ist im Skript detailliert beschrieben.

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 $\text{⌘} + \text{→}$ oder $\text{⌘} + \text{←}$ in die Landscape Orientierung, werden die Frames der einzelnen Views nicht verändert und die Benutzeroberfläche wird nicht wie gewünscht angezeigt (s. S. 30, Abb. 4.1).

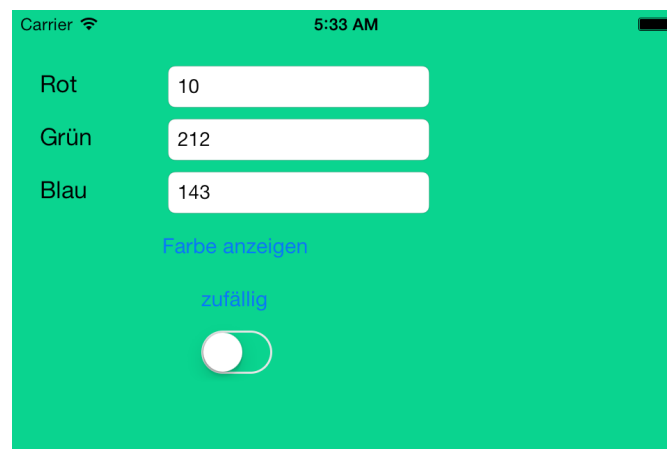


Abbildung 4.1: In der Landscape Orientierung bleiben die Frames einfach erhalten

2. Um das Problem zu lösen, können wir nach dem Auto Layout Konzept nun Constraints definieren und damit **NSLayoutConstraint** Objekte einer View 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.

3. 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.
4. 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.
5. 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 dabei die **Intrinsic Content Size**!

In einem eindeutigen Layout werden die Constraints blau markiert und ihr könnt das *Resolve Auto Layout Issues* Menü verwenden, um mit einem Klick auf *Update All Frames in View Controller* alle Views entsprechend ihrer Constraints zu positionieren. Bei einem eindeutigen Layout werden die Frames anschließend automatisch angepasst, wenn ihr die Konstanten der Constraints verändert.

6. Wählt ihr im Interface Builder das *View Controller* Objekt aus, so könnt ihr im Attributes Inspector `Orientation > Landscape` wählen. Die View wird dann mit entsprechend angepasstem Frame angezeigt und ihr könnt die Funktionsweise eurer Constraints testen.

Übungsaufgaben

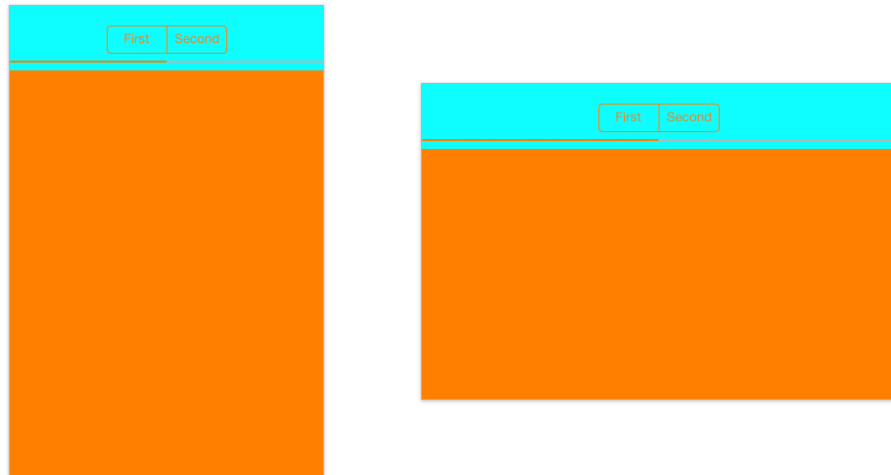
9. Auto Layout

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.

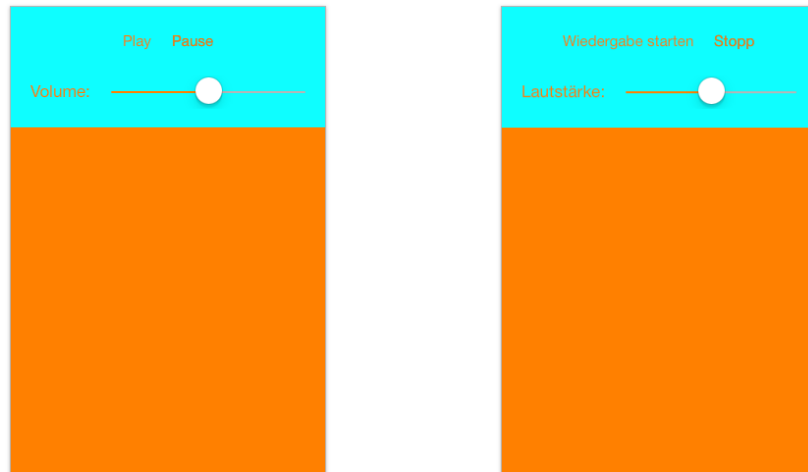
Löst dann die folgenden Layouts durch geschickte Definition von Constraints. Ihr könnt einem beliebigen Storyboard einfach für jedes Problem ein *View Controller* Objekt aus der Object Library hinzufügen und dessen View Hierarchie im Storyboard mit Views und Constraints konfigurieren.

Hinweis: 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.

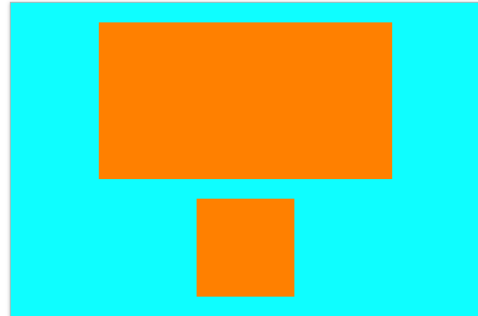
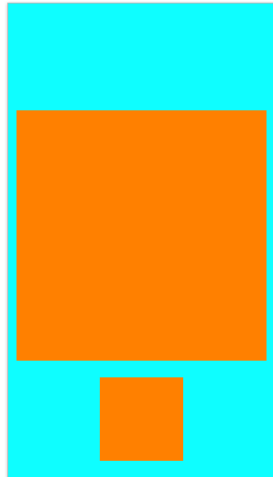
- a) Eine `UISegmentedControl` und eine `UIProgressView` sind am oberen Bildschirmrand positioniert, eine `UIView` füllt den verbliebenen Platz.



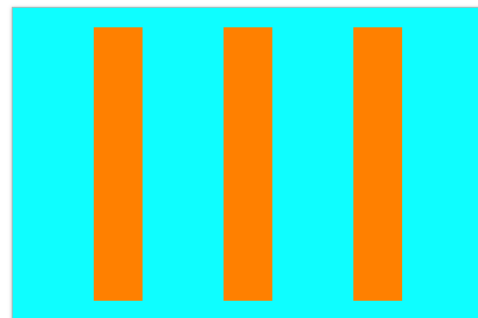
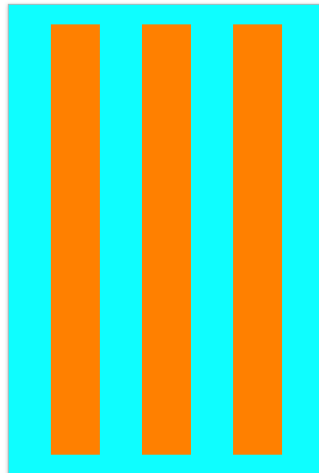
- b) Zwei Buttons im Abstand von 20pt sind am oberen Bildschirmrand zusammen horizontal zentriert. Ein **UISlider** mit zugehörigem Label und eine füllende View befinden sich darunter. Ändern wir den Text der Label, passt sich das Layout an.



- 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. 28, Übungsaufgabe 8). 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 Button 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 `ViewController` : `UIViewController` in unserem Projekt.
2. Die Klasse `ViewController` möchten wir zunächst in `CitiesViewController` umbenennen. Dazu verwenden wir Xcode's **Refactor** Funktion. Markiert den Namen der Klasse an einer beliebigen Stelle im Code und wählt mit einem Rechtsklick im Kontextmenü `Refactor` > `Rename...` (s. S. 35, Abb. 5.1). Geben wir nun den Namen

CitiesViewController ein und bestätigen den Dialog, benennt Xcode das Symbol an allen Stellen im Code, in Storyboards und in Dateinamen um.

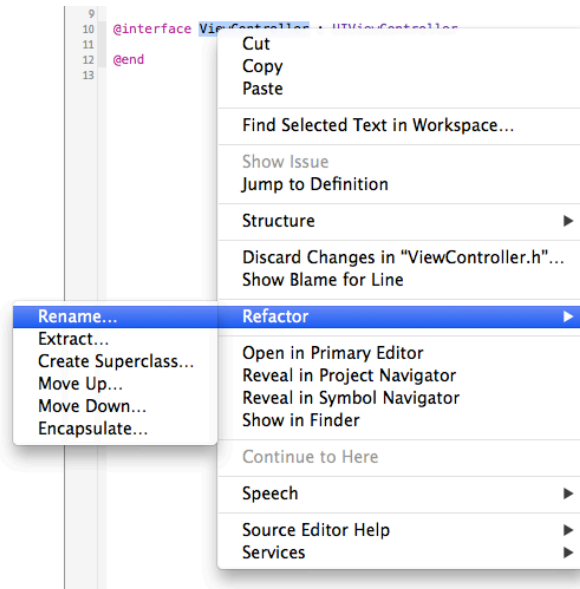


Abbildung 5.1: Die Refactor Funktion ist sinnvoll, um Symbole im gesamten Projekt umzubenennen

3. Im Storyboard finden wir eine Scene mit einem Objekt dieser (nun umbenannten) CitiesViewController Klasse. Wie im Skript dargestellt, können wir uns im Identity Inspector davon überzeugen, dass die Klasse des View Controller Objekts tatsächlich CitiesViewController ist. Zur Laufzeit wird also ein solches Objekt 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 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. 36, Abb. 5.2). Als Titel des Buttons könnt ihr auch zunächst generisch *City name* o.ä. wählen, da wir diesen im Code anpassen.

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 eine neue Klasse City : NSObject und definiert die Attribute NSString*name und UIImage*image.

```
1 // City.h
```



Abbildung 5.2: Ein Button soll bei Betätigung Informationen über die entsprechende Stadt anzeigen

```

2
3 @interface City : NSObject
4
5 @property (strong, nonatomic) NSString *name;
6 @property (strong, nonatomic) UIImage *image;
7
8 @end

```

```

1 // City.m
2
3 #import "City.h"
4
5 @implementation City
6
7 @end

```

6. Später wird der Cities View Controller eine Liste von Städten anzeigen, doch zunächst beschränken wir uns auf eine. Fügt daher ein Attribut `City *city` im öffentlichen Interface hinzu.

```

1 // CitiesViewController.h
2
3 @class City; // Forward Declaration
4
5 @interface CitiesViewController : UIViewController
6
7 @property (strong, nonatomic) City *city;
8
9 @end

```

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 weiterzuge-

ben. Implementiert daher folgende `application:didFinishLaunchingWithOptions:` Methode.

```

1 // AppDelegate.m
2
3 #import "AppDelegate.h"
4 #import "City.h"
5 #import "CitiesViewController.h"
6
7 @implementation AppDelegate
8
9 - (BOOL)application:(UIApplication *)application
10   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
11 {
12     City *city = [[City alloc] init];
13     city.name = @"Melbourne";
14     city.image = [UIImage imageNamed:@"melbourne"];
15
16     CitiesViewController *citiesVC = self.window.rootViewController;
17     citiesVC.city = city;
18
19     return YES;
20 }
21 @end

```

Es gibt natürlich auch Städte in Mitteleuropa, Panama und auf Neuseeland...

Hinweis: Die Klassenmethode `imageNamed:` von `UIImage` lädt die Bilddatei mit dem angegebenen Dateinamen, sofern die Datei im Target referenziert ist. Für Dateien im PNG Format muss die Dateierweiterung nicht enthalten sein. 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 Dateierweiterung `.xassets` ist bereits im Projekt enthalten. Auch dieser Datei könnt ihr per *Drag & Drop* Bilddateien hinzufügen.

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

```

1 // CitiesViewController.m
2
3 #import "CitiesViewController.h"
4 #import "City.h"
5
6 @interface CitiesViewController ()
7
8 @property (strong, nonatomic) IBOutlet UIButton *cityButton;
9
10 @end
11
12 @implementation CitiesViewController
13
14 - (void)viewWillAppear:(BOOL)animated
15 {
16     [super viewWillAppear:animated];
17 }
18
19 @end

```

```

17
18     [self.cityButton setTitle:self.city.name forState:UIControlStateNormal
19     ];
20 }
21 @end

```

9. Um nun 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 `City *city` im Header der `CityDetailViewController` Klasse.

```

1 // CityDetailViewController.h
2
3 @class City; // Forward Declaration
4
5 @interface CityDetailViewController : UIViewController
6
7     @property (strong, nonatomic) City *city;
8
9 @end

```

10. Nun verwenden wir wieder 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 nun im Identity Inspector des hinzugefügten View Controllers 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. 39, Abb. 5.3). Fügt außerdem einen *Zurück* Button hinzu.

```

1 // CityDetailViewController.m
2
3 #import "CityDetailViewController.h"
4
5 @interface CityDetailViewController ()
6
7     @property (strong, nonatomic) IBOutlet UILabel *nameLabel;
8     @property (strong, nonatomic) IBOutlet UIImageView *imageView;
9
10 @end

```

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. 40, Abb. 5.4). Erstellt so eine **Modal Segue** zwischen Button und View Controller.

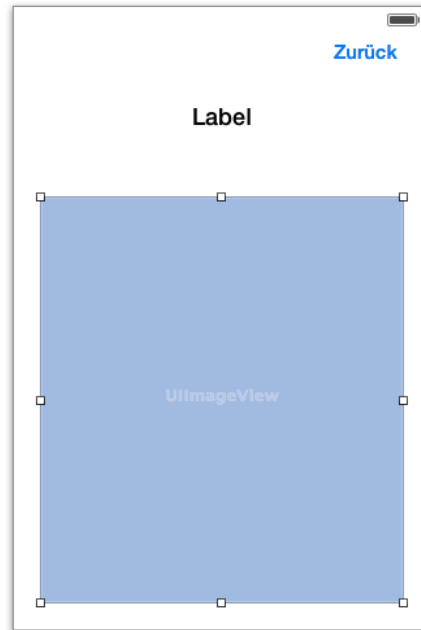


Abbildung 5.3: Der City Detail View Controller soll Information über die ausgewählte Stadt anzeigen

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.m
2
3 #import "CitiesViewController.h"
4 #import "City.h"
5 #import "CityDetailViewController.h"
6
7 @interface CitiesViewController ()
8
9 @property (strong, nonatomic) IBOutlet UIButton *cityButton;
10
11 @end
12
13 @implementation CitiesViewController
14
15 - (void)viewWillAppear:(BOOL)animated
16 {
17     [super viewWillAppear:animated];
18
19     [self.cityButton setTitle:self.city.name forState:UIControlStateNormal];
20 }

```

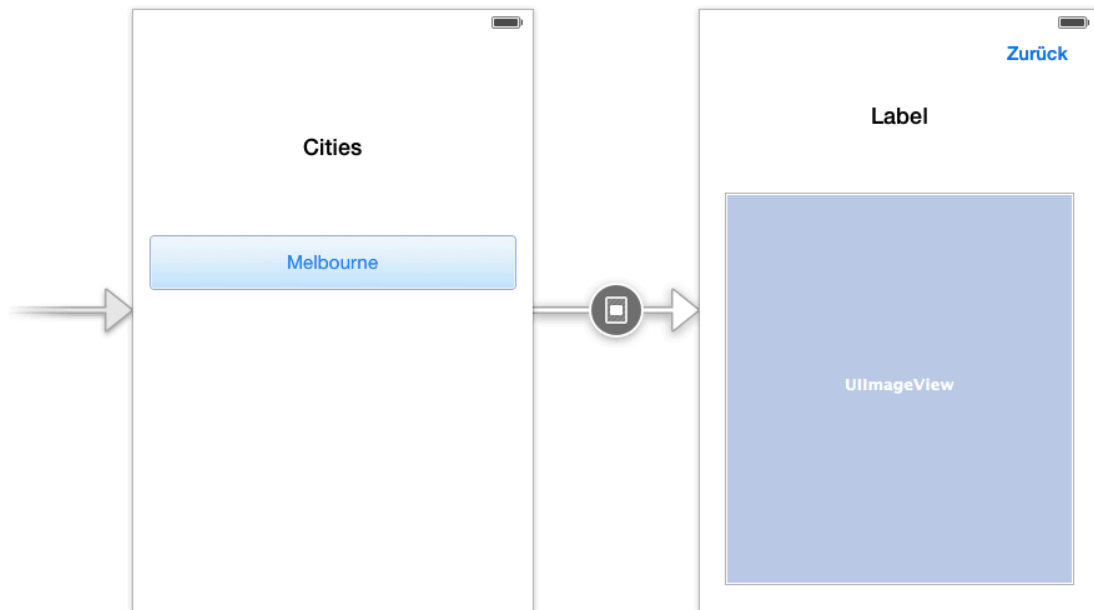


Abbildung 5.4: Eine Modal Segue konfiguriert die Präsentation des City Detail View Controllers bei Betätigung des Buttons

```

20 }
21
22 - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
23 {
24     if ([segue.identifier isEqualToString:@"showCityDetail"]) {
25         CityDetailViewController *cityDetailVC = segue.
            destinationViewController;
26         cityDetailVC.city = self.city;
27     }
28 }
29
30 @end

```

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 weiteren Implementierung der `viewWillAppear:` Instanzmethode in der `CityDetailViewController` Klasse.

```

1 #import "CityDetailViewController.h"
2 #import "City.h"
3
4 @interface CityDetailViewController ()
5
6 @property (strong, nonatomic) IBOutlet UILabel *nameLabel;
7 @property (strong, nonatomic) IBOutlet UIImageView *imageView;
8

```



```

9  @end
10
11  @implementation CityDetailViewController
12
13  - (void)viewWillAppear:(BOOL)animated
14  {
15      [super viewWillAppear:animated];
16
17      self.nameLabel.text = self.city.name;
18      self.imageView.image = self.city.image;
19  }
20
21  @end

```

Betätigt ihr nun den Button, wird die Content View des City Detail View Controller angezeigt und mit dem entsprechenden City Objekt konfiguriert (s. S. 41, Abb. 5.5).

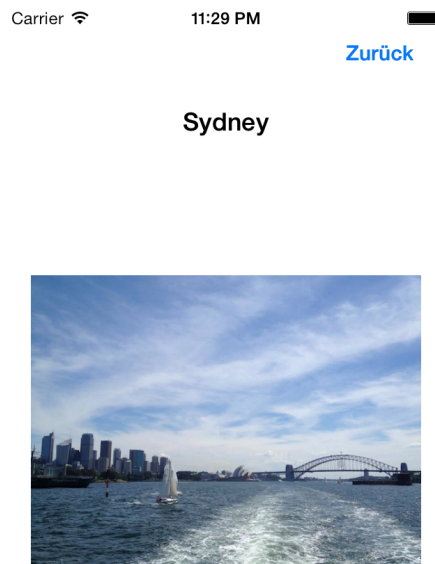


Abbildung 5.5: Wird der Button betätigt, zeigt der City Detail View Controller die Informationen zu der ausgewählten Stadt

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.m
2
3  #import "CitiesViewController.h"
4  #import "City.h"
5  #import "CityDetailViewController.h"
6
7  @interface CitiesViewController ()
8
9  - (IBAction)unwindToCities:(UIStoryboardSegue *)segue;
10
11 @end
12
13 @implementation CitiesViewController
14
15 - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
16 {
17     if ([segue.identifier isEqualToString:@"showCityDetail"]) {
18         City *city = [[City alloc] init];
19         city.name = @"Melbourne";
20         city.image = [UIImage imageNamed:@"melbourne"];
21         CityDetailViewController *cityDetailVC = segue.
22             destinationViewController;
23         cityDetailVC.city = city;
24     }
25 }
26
27 - (IBAction)unwindToCities:(UIStoryboardSegue *)segue
28 {
29     // no implementation necessary
30 }
31 @end

```

16. Im Storyboard können wir nun das `UIControlEventTouchUpInside` 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. 43, Abb. 5.6). 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

10. Cities

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

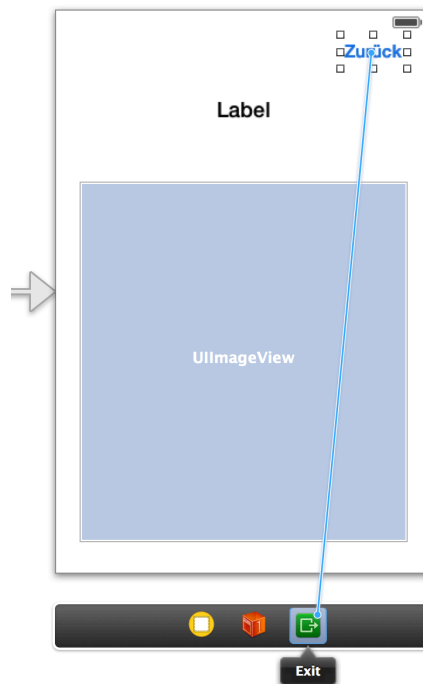


Abbildung 5.6: Die *Exit* Schaltfläche präsentiert alle Unwind Segues in der View Controller Hierarchie

5.2 One City Navigation

UIKit stellt für diese häufig verwendete Master-Detail View Controller Hierarchie die Subklasse `UINavigationController`: `UIViewController` zur Verfügung. 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. 44, Abb. 5.7).
2. Ihr könnt nun die Modal Segue zwischen `CitiesViewController` und `CityDetailViewController` auswählen und im Attributes Inspector den Typ zu **Push Segue** ändern. Nun wird automatisch statt der `presentViewController:animated:completion` Methode von `UIViewController` die `pushViewController:animated:` Methode von `UINavigationController` aufgerufen, wenn die Segue ausgelöst wird.
3. Den *Zurück* Button könnt ihr nun entfernen, da `UINavigationController` einen eigenen Mechanismus implementiert und eine `UINavigationBar` am oberen Bildschirm-

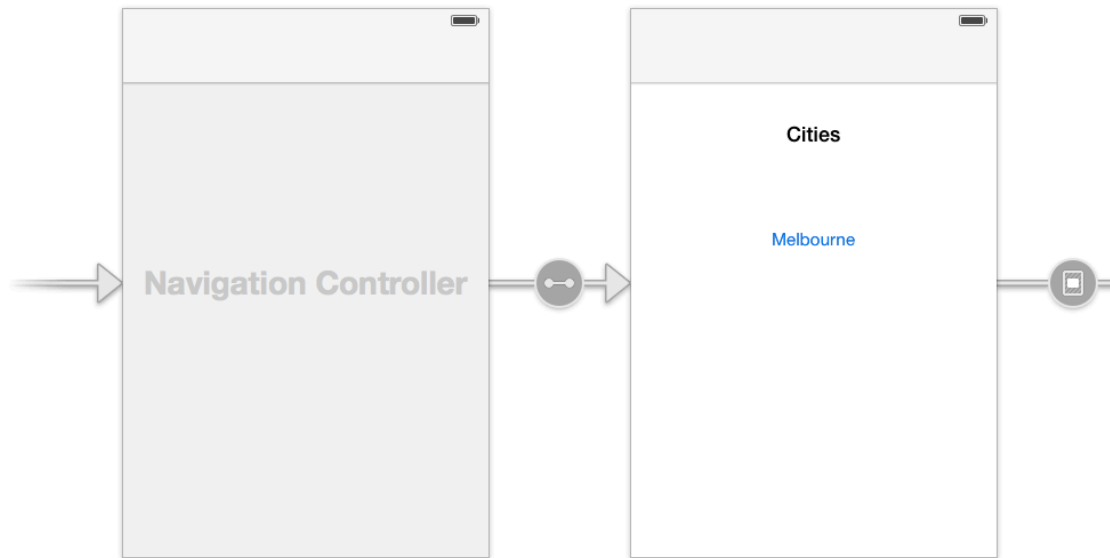


Abbildung 5.7: Eine Relationship Segue markiert den Root View Controller eines Navigation Controllers

rand anzeigt. Auch die Labels werden nicht mehr benötigt. Stattdessen können wir das `NSString*title` Attribut von `UIViewController` verwenden, dessen Wert als Titel in der Navigation Bar angezeigt wird (s. S. 45, Abb. 5.8).

```

1 // CityDetailViewController.m
2
3 #import "CityDetailViewController.h"
4 #import "City.h"
5
6 @interface CityDetailViewController ()
7
8 @property (strong, nonatomic) IBOutlet UIImageView *imageView;
9
10 @end
11
12 @implementation CityDetailViewController
13
14 - (void)viewWillAppear:(BOOL)animated {
15     [super viewWillAppear:animated];
16     self.navigationItem.title = self.city.name;
17     self.imageView.image = self.city.image;
18 }
19
20 @end

```

Verwendet außerdem gerne Auto Layout, um die Views zu positionieren.

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.

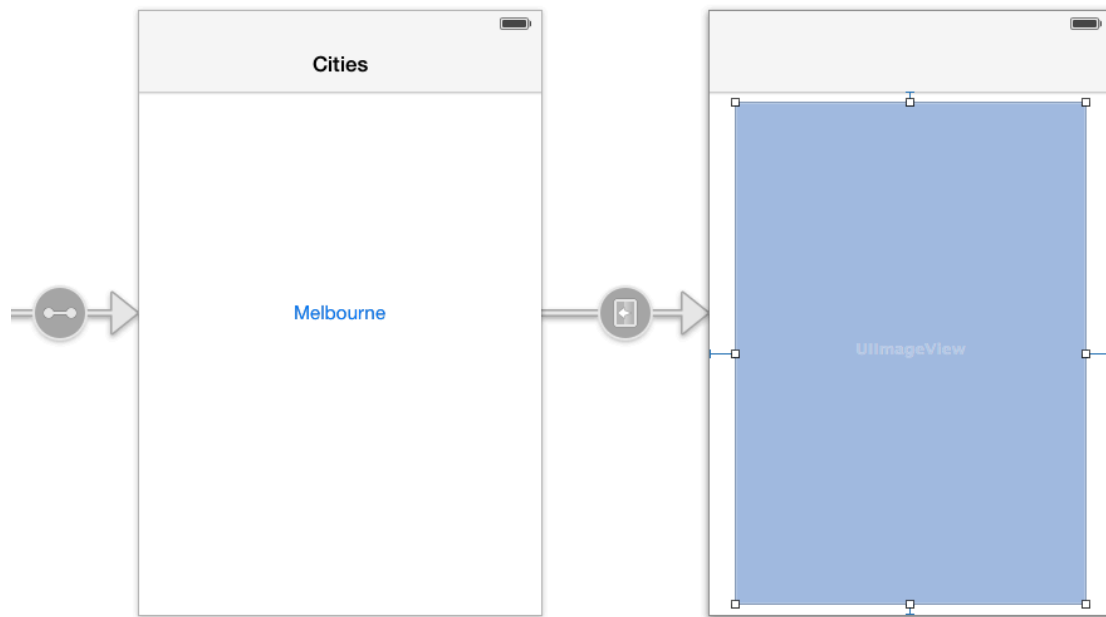


Abbildung 5.8: Navigation Controller zeigen eine Navigation Bar an

```

1 // AppDelegate.m > application:didFinishLaunchingWithOptions:
2 CitiesViewController *citiesVC = (CitiesViewController *)([
    UINavigationController *)self.window.rootViewController
    topViewController];

```

5. Durch die Verwendung des Navigation Controllers erhält unsere App nun die Standardanimationen und -mechanismen aus UIKit, die sich bspw. durch die Verwendung von Subklassen und sog. Delegates vielseitig anpassen lassen (s. S. 46, Abb. 5.9).

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 `cityButton` 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



Abbildung 5.9: Navigation Controller stellen Standardanimationen und -mechanismen zur Verfügung

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. 47, Abb. 5.10).
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

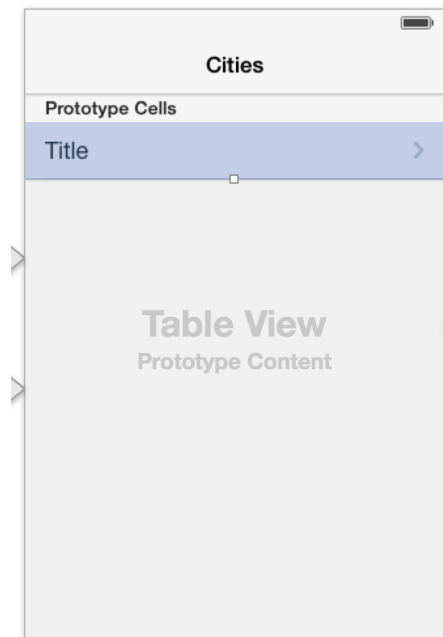


Abbildung 5.10: Prototype Cells dienen als Vorlage für die Zellen der Table View

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 `id<UITableViewDataSource>datasource` und `id<UITableViewDelegate>delegate` Objekte zuzuweisen, sind diese als IBOutlets markiert. Zieht also zwei Verbindungen von der Table View zum `CitiesViewController` und wählt diesen damit sowohl als Delegate als auch als DataSource.
7. Im Allgemeinen repräsentiert eine Table View ein `NSArray`. Definiert also im öffentlichen Interface des `CitiesViewController` ein Attribut `NSArray*cities` anstatt des bisherigen `City *city` Attributs.

```

1 // CitiesViewController.h
2
3 @interface CitiesViewController : UITableViewController
4
5 @property (strong, nonatomic) NSArray *cities;
6
7 @end

```

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

```

1 // AppDelegate.m
2
3 - (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
4 {
5     // create some city objects
6     City *melbourne = [[City alloc] init];
7     melbourne.name = @"Melbourne";
8     melbourne.image = [UIImage imageNamed:@"melbourne"];
9     City *sydney = [[City alloc] init];
10    sydney.name = @"Sydney";
11    sydney.image = [UIImage imageNamed:@"sydney"];
12    // (...)
13
14    // provide root view controller with model objects
15    CitiesViewController *citiesVC = (CitiesViewController *)([
        UINavigationController *)self.window.rootViewController
        topViewController];
16    citiesVC.cities = @[melbourne, sydney];
17
18    return YES;
19 }

```

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 #pragma mark - Table View Datasource
2
3 - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
4     return 1;
5 }
6
7 - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(
    NSInteger)section {
8     return [self.cities count];
9 }
10
11 - (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
12
13     // Greift auf das Model zurück, um das darzustellende Objekt zu
        erhalten
14     City *city = [self.cities objectAtIndex:indexPath.row];
15
16     // Erstellt ein neues UITableViewCell Objekt entsprechend der im
        Storyboard definierten Prototype Cell mit dem gegebenen
        Identifier oder verwendet eine existierende, momentan nicht
        verwendete Zelle mit diesem Identifier
17     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"
        cityCell" forIndexPath:indexPath];
18
19     // Konfiguriert die View entsprechend des Models
20     cell.textLabel.text = city.name;
21     cell.imageView.image = city.image;

```



```

22
23     return cell;
24 }

```

`#pragma mark` kennzeichnet eine Überschrift ähnlich eines Kommentars und wird zusätzlich in der Jump Bar angezeigt. Ein Bindestrich – erzeugt dort eine horizontale Trennlinie.

Die zuvor implementierten Methoden werden nicht mehr benötigt.

9. Diese drei Methoden des `UITableViewDataSource` Protokolls stellen der Table View die erforderlichen Informationen zur Verfügung, um die Tabelle anzuzeigen (s. S. 49, Abb. 5.11).

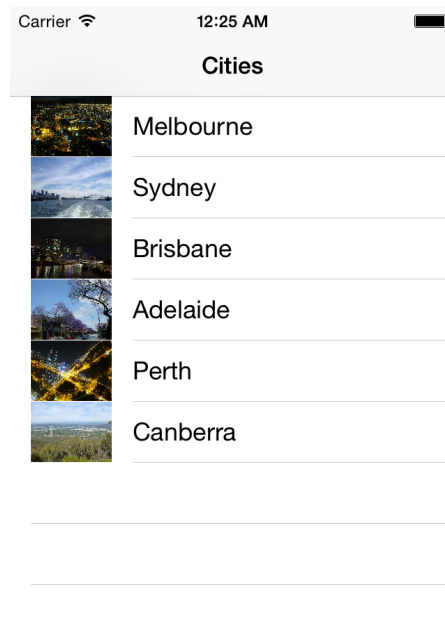


Abbildung 5.11: Eine Table View stellt Anfragen an ihr Datasource und Delegate Objekt, um Zellen dynamisch anzuzeigen

10. Die Präsentation des Detail City View Controllers kann nun erneut mithilfe von Segues realisiert werden. Dazu können wir im Storyboard eine Push 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 zunächst den Index Path der betätigten Zelle herausfinden. Damit lässt sich anschließend das entsprechende Model Objekt erhalten und der City Detail View Controller konfigurieren.

```

1 // ViewController.m
2

```

```

3  #import "CitiesViewController.h"
4  #import "City.h"
5  #import "CityDetailViewController.h"
6
7  @implementation CitiesViewController
8
9  #pragma mark – User Interaction
10
11  – (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
12  {
13      if ([segue.identifier isEqualToString:@"showCityDetail"]) {
14
15          City *city = [self.cities objectAtIndex:self.tableView.
16                      indexPathForSelectedRow.row];
17
18          CityDetailViewController *cityDetailVC = segue.
19              destinationViewController;
20          cityDetailVC.city = city;
21      }
22  }
23
24  #pragma mark – Table View Datasource
25
26  // (...)
27
28  @end

```

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

11. Cities 2

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