iPhone- und iPad-Programmierung für Einsteiger

iOS-Apps entwickeln von Anfang an

INGO BÖHME

Markt+Technik

ALWAYS LEARNING

PEARSON

iPhone- und iPad-
Programmierung für Einsteiger

Ingo Böhme

iPhone- und iPad-
Programmierung für Einsteiger

iOS-Apps entwickeln von Anfang an

Markt+Technik

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

10 9 8 7 6 5 4 3 2

14 13 12

ISBN 978-3-8272-4713-1

ein Imprint der Pearson Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Einbandgestaltung: Marco Lindenbeck, webwo GmbH, mlindenbeck@webwo.de
Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Herstellung: Elisabeth Prümm, epruemm@pearson.de
Korrektorat: Brigitte Hamerski, Willich
Satz: Nadine Krumm, mediaService, Siegen (www.media-service.tv)
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

Kapitel 3

Objective-C – objektorientiert

Nachdem im zweiten Kapitel die Grundlagen für die Sprache C gelegt worden sind, kommt nun der interessante Teil. Denn erst mit der Objektorientierung wird der Teil von Objective-C wirklich genutzt, der in letzter Konsequenz so fasziniert. Und damit auch der Teil, der bei der ersten Begegnung so kompliziert wirkt.

3.1 Erste Schritte im Interface Builder

Weniger in PHP, dafür umso häufiger in VisualBasic und Delphi arbeitet man in einer Entwicklungsumgebung, bei der die Gestaltung des Benutzerinterface per Drag&Drop zusammengeklickt wird. Dann werden Eigenschaften gesetzt und mithilfe von Methoden die einzelnen Objekte, von der Schaltfläche bis hin zum Label, mit Leben gefüllt. Im Grunde genommen ist es bei Objective-C genauso. Der einzige Unterschied ist, dass es bei VisualBasic und Delphi ganz starre Formen der Notation gibt. In Objective-C hingegen legt der Programmierer Instanzenvariablen und die Namen der Methoden selbst fest – etwa den Namen jener Methode, die bei einer Schaltfläche beim Klick-Ereignis ausgeführt werden soll. Am einfachsten sehen Sie dies an einem kleinen Beispiel. Daher werden wir in den folgenden Abschnitten peu à peu visuell eine kleine Applikation zusammenbasteln, die entsprechenden wichtigen Stellen im Quellcode aufsuchen und dann die Elemente mit Leben füllen. Während bei VisualBasic und Delphi die visuelle Gestaltung und die Programmierung in derselben Oberfläche stattfinden, gibt es dafür in Objective-C zwei Tools:

- *Xcode*, um – wie der Name schon sagt – den Code festzulegen und
- *Interface Builder*, der auch das tut, was sein Name vermuten lässt: Er hilft nämlich, das Interface, also die Benutzeroberfläche der iPhone-Applikation zu gestalten.

Seit der Version 4 von Xcode sind diese beiden Tools aber zumindest unter einer Oberfläche zusammengefasst. Und glauben Sie mir: Das ständige Hin- und Herschalten werden die Neueinsteiger in die Version 4 sicher nicht vermissen. Die Aufsteiger von der Version 3 hingegen haben allen Grund zur Freude.

Um das Ganze an einem Beispiel kennenzulernen, starten Sie Xcode neu. Legen Sie ein iPhone-Application-Projekt vom Typ Single View Application an. Nennen Sie dieses *Kapitel3*.

Projektquellcode

Sie finden den Quellcode des Projekts unter *www.iho.me/Kapitel3*.

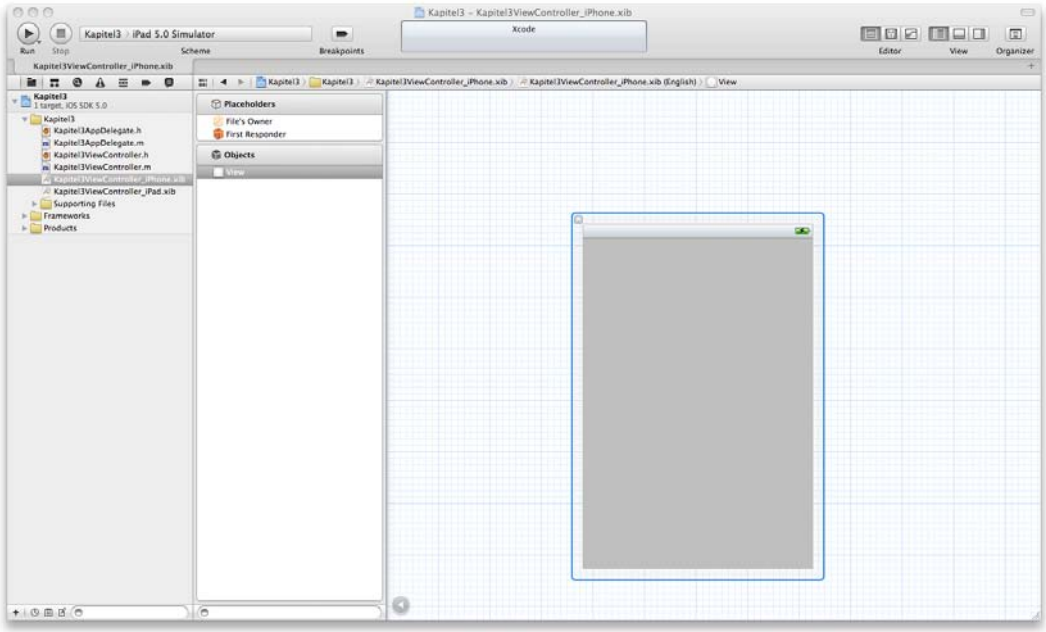

Abbildung 3.1: Hinter den Dateien mit der Endung .xib verstecken sich die Interface-Builder-Layouts.

In der Projektübersicht sehen Sie den Eintrag *Kapitel3ViewController.xib*. Eventuell sehen Sie, wenn Sie als Projektvorlage UNIVERSAL für iPhone und iPad gewählt haben, noch die Endungen des entsprechenden iDevice.

Diese *.xib*-Dateien (sprich: „sipp") sind Interface-Builder-Dateien, also jene Dateien, in denen das Layout gespeichert wird. Doppelklicken Sie auf diesen Eintrag, zeigt sich der Interface Builder. Standardmäßig ist das Fenster in drei vertikal getrennte Bereiche geteilt.

Das VIEW-Fenster an sich entspricht beim *SINGLE VIEW APPLICATION*-Projekt-Template der Fensteransicht. Hier werden in einem Fenster ein oder mehrere Views angezeigt. Dies ist also der Platz, in dem die einzelnen Elemente – von der Schaltfläche über Labels bis hin zu Textfeldern oder Bildern – angeordnet werden. Zu diesem Fenster gehört ein weiteres, das den Namen der *.xib*-Datei trägt, also in unserem Fall *Kapitel3ViewController.xib*.

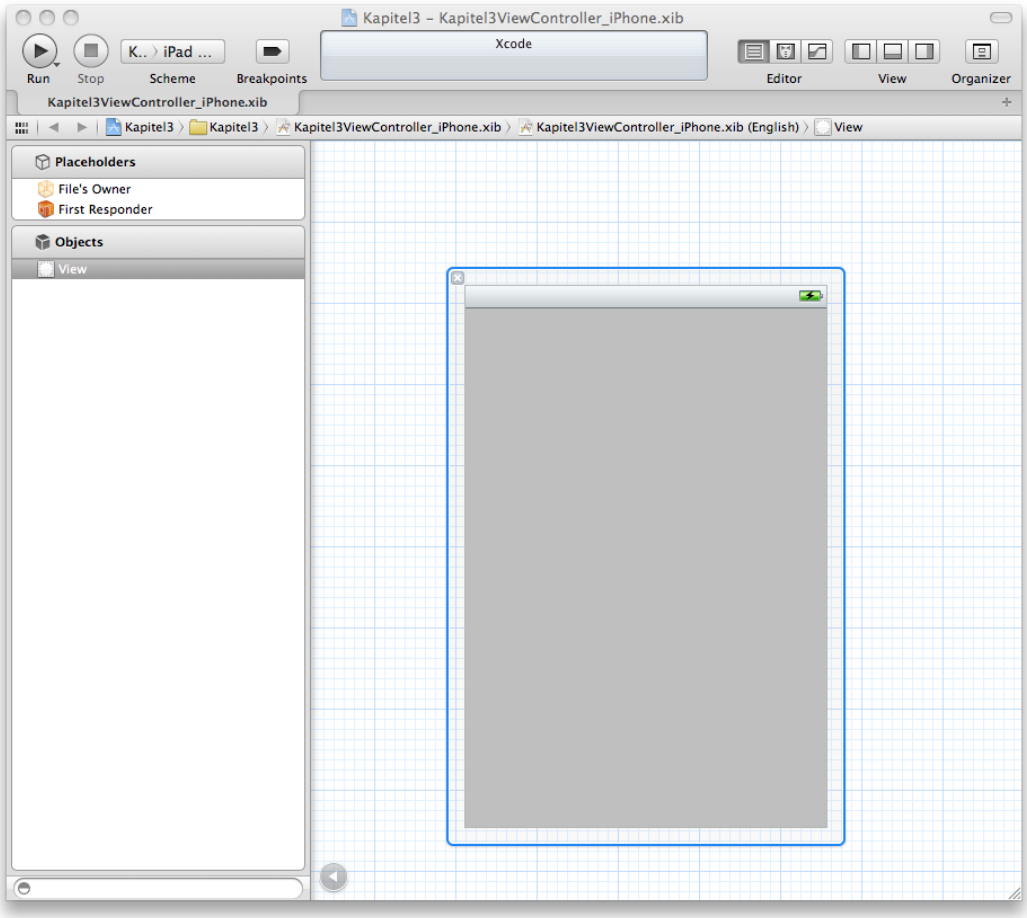

Abbildung 3.2: Das View-Fenster (rechts) stellt die Grundlage der Ansicht dar und wird später mit Steuerelementen gefüllt und gestaltet.

Das Object Library-Fenster (Ctrl + ⌥ + ⌘ + 3) enthält verschiedene sichtbare und unsichtbare Steuerelemente, die der Cocoa Touch-Programmierer verwenden kann, um im Interface Builder die Oberfläche zu gestalten. Diese Steuerelemente sind aufeinanderfolgend in Kategorien geordnet: Erst kommen die Schaltflächen und andere typische User-Interface–Elemente, und die Auswahl reicht bis zu Rahmen für Bilder oder HTML-Container.

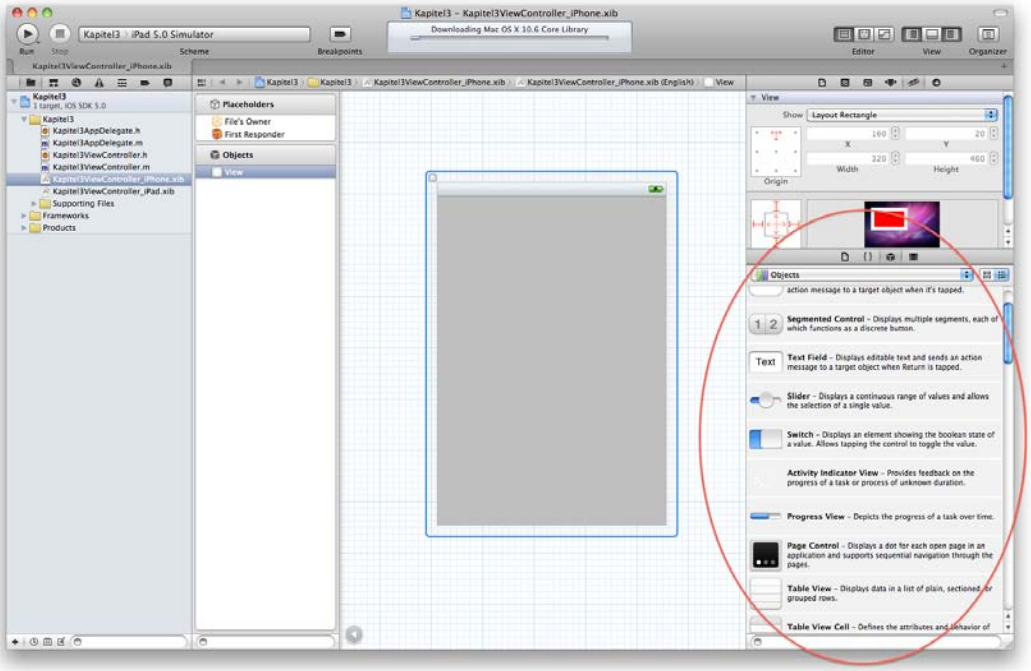

Abbildung 3.3: Am rechten Rand, im Object Library-Fenster des Interface Builders, sind alle sichtbaren und unsichtbaren Steuerelemente enthalten.

Der Inspector – der obere Teil der rechten Bildschirmspalte – ist dazu da, einzelne Elemente zu untersuchen, Text, Farbe und Erscheinen zu formatieren und Eigenschaften im Quelltext mit den Elementen im Interface Builder zu verbinden. Hier werden auch die Verknüpfungen zwischen den Methoden im Objective-C-Code und den Steuerelementen der gestalteten Oberfläche hergestellt. Der Inspector ist in fünf Register eingeteilt, die durch die fünf unterschiedlichen Symbole über dem Fensterbereich rechts dargestellt sind.

Das erste Register von links ist der File Inspector (⌥+⌘+1). In dieser Ansicht sehen Sie Informationen über die aktuell geöffnete *.xib*-Datei sowie deren Lokalisierung. In Xcode 4 ist es sehr leicht, Anwendungen für unterschiedliche Märkte zu schreiben. Alles, was Sie machen müssen, ist, eine *.xib*-Datei für die unterschiedlichen Lokalisierungen, wie beispielsweise für Frankreich, USA, Spanien usw., im Interface Builder zu erstellen. Die verschiedenen Varianten sind tatsächlich in einer einzigen *.xib*-Datei enthalten, und das iPhone oder das iPad verwendet einfach jene, die am besten zum gewählten Lokalisierungsprofil des iDevice passt.

Das zweite Register ist der sogenannte Quick Help Inspector (⌥+⌘+2). Wenn Sie ein Element oder das View selbst markieren, erhalten Sie eine ausführliche Beschreibung samt der Querverweise in die Xcode-Hilfe. Besonders beachtenswert ist der Abschnitt Sample Code. Hier können Sie – wenn Sie etwas weiter im Buch fortgeschritten sind – die Arbeitsweise des jeweiligen Steuerelements in der Praxis (sprich: im Objective-C Code) sehen.

Der Identity Inspector (⌥+⌘+3) des dritten Registers hilft Ihnen dabei, das jeweilige Steuerelement mit der zugehörigen Quellcodedatei, genauer gesagt: mit dem zugehörigen Klassenmodul, zu verbinden. Im Gegensatz beispielsweise zu Delphi oder Visual Basic können nämlich Eigenschaften und Methoden verschiedener Steuerelemente auf einem einzigen Fensterobjekt in unterschiedlichen Quellcodedateien abgearbeitet werden.

Das vierte Register ist auch gleichsam das wichtigste für die gestalterische Arbeit. Im Attributes Inspector (⌥+⌘+4) legen Sie die Eigenschaften des jeweils markierten Steuerelements oder des Fensters fest. Markieren Sie eines der Elemente in der Attributes Library, beispielsweise ein Label, und ziehen Sie es auf die View-Fläche. Ist es markiert, so sehen Sie im Object Inspector sämtliche Attribute des ausgewählten Elements.

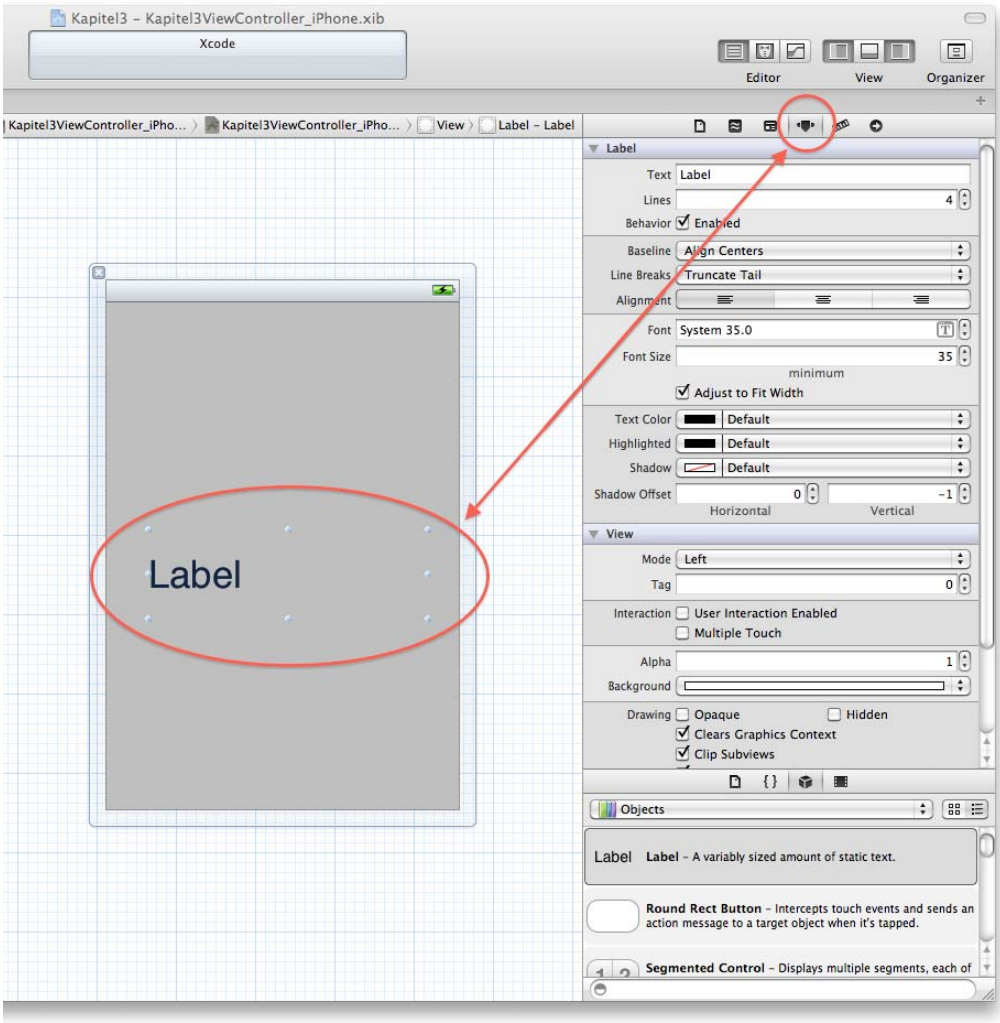

Abbildung 3.4: Der Inspector setzt Eigenschaften von Views und Steuerelementen.

Register Nummer fünf, der Size Inspector (⌥+⌘+5), hilft Ihnen dabei, die Größe und Position der Steuerelemente visuell festzulegen und ihr Verhalten beim Verändern der Display-Größe zu beschreiben. Eine Veränderung des Displays tritt beispielsweise auf, wenn Sie das iDevice um 90° drehen. Dann können Sie im unteren Teil des Size Inspector festlegen, wo das Steuerelement nach der Größenänderung dargestellt werden soll und ob es sich eventuell auch in Höhe und Breite verändert.

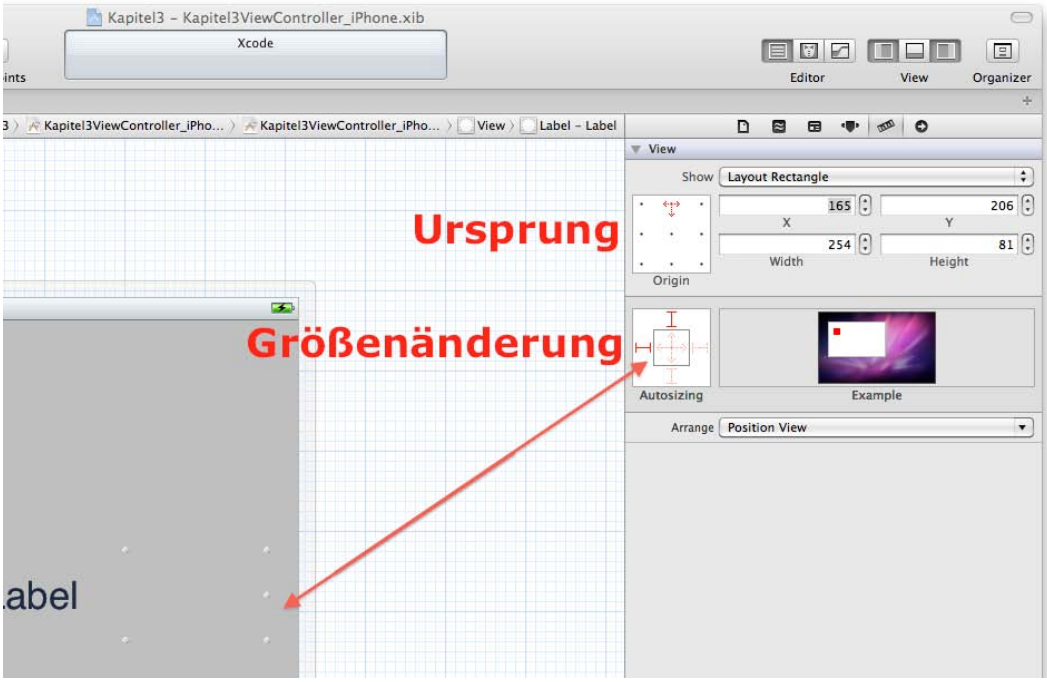

Abbildung 3.5: Im Size Inspector legen Sie das Verhalten von Views und Steuerelementen fest, sobald sich das iDevice dreht.

Im sechsten und letzten Register, dem Connection Inspector, sehen Sie die Verbindungen zwischen den einzelnen Steuerelementen im gestalteten Formular und dem dazugehörigen Objective-C-Quellcode. Hier werden zum einen die Namen festgelegt, über die im Code auf die Steuerelemente zugegriffen wird – etwa um den Text eines Labels zu ändern –, und zum anderen finden Sie hier den direkten Zusammenhang zwischen Ereignissen der verschiedenen Steuerelemente und den Methoden, die Sie in Objective-C für diese Ereignisse zur Verfügung stellen können.

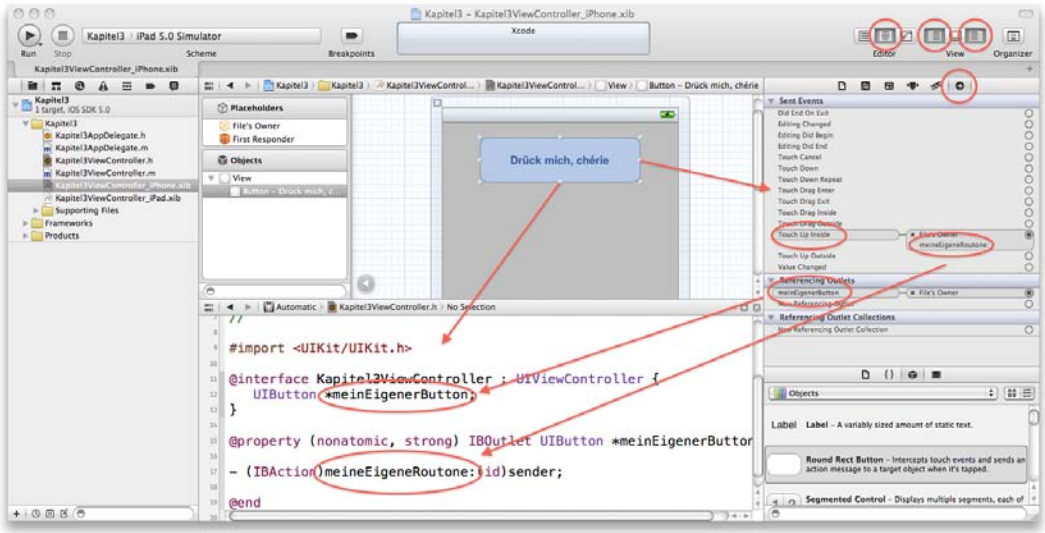

Abbildung 3.6: Im Connection Inspector *legen Sie die Verbindungen zwischen dem visuellen Gestaltungstool Interface Builder und dem Objective-C-Quellcode fest.*

3.2 Steuerelemente platzieren und ausrichten

Als Erstes setzen Sie drei Steuerelemente auf das Form, das Formular, das Fenster oder (wie es eben in Objective-C heißt) das *View*. Schauen Sie dazu in der Objektbibliothek nach einem Element namens Round Rect Button. Ziehen Sie es dann per Drag&Drop auf das View-Formular. Bewegen Sie das Steuerelement in Richtung der unteren linken Ecke. Wenn die Schaltfläche einen vernünftigen Abstand vom Rand links und unten hat, erscheinen Hilfslinien, die es Ihnen erleichtern, das Steuerelement vernünftig zu platzieren. Lassen Sie die Maustaste los, sobald die Schaltfläche links unten an den Hilfslinien ausgerichtet ist. Ist das Steuerelement links unten platziert, bekommt es die von Grafikprogrammen bekannten acht Anfasser an den Ecken und Seiten, mit denen die Größe verändert werden kann. Ziehen Sie es am rechten seitlichen Anfasser in die Breite, bis auf der rechten Seite die Hilfslinie erscheint. Lassen Sie die Schaltfläche dann erneut los, ist sie perfekt am unteren Rand des sichtbaren Bereichs angeordnet. Wenn Sie nun einen Doppelklick auf der Schaltfläche ausführen, können Sie die Beschriftung der Schaltfläche direkt eingeben. Tragen wir beispielsweise *Drück mich!* ein.

Alle weiteren Attribute ändern Sie im *Inspector* rechts oben in der rechten Fensterspalte.

Hinweis

Die rechte Spalte mit dem Inspector sowie der Objektbibliothek sehen Sie nur, wenn Sie oben rechts im Titel des Fensters bei den Symbolen im Abschnitt VIEW auch die rechte Spalte markiert haben.

Achten Sie darauf, dass die Schaltfläche im View noch markiert ist. Probieren Sie ruhig ein wenig mit den Eigenschaften im ATTRIBUTES INSPECTOR (⌥+⌘+4) herum. Wenn Sie es am Ende nicht mehr schaffen, den ursprünglichen Status der Schaltfläche wiederherzustellen – was soll's! Es ist ja so leicht, mithilfe der Gestaltungsoberfläche, die fast an ein Grafikprogramm wie CorelDRAW, Photoshop oder Freehand erinnert, einfach eine neue Schaltfläche aufzuziehen.

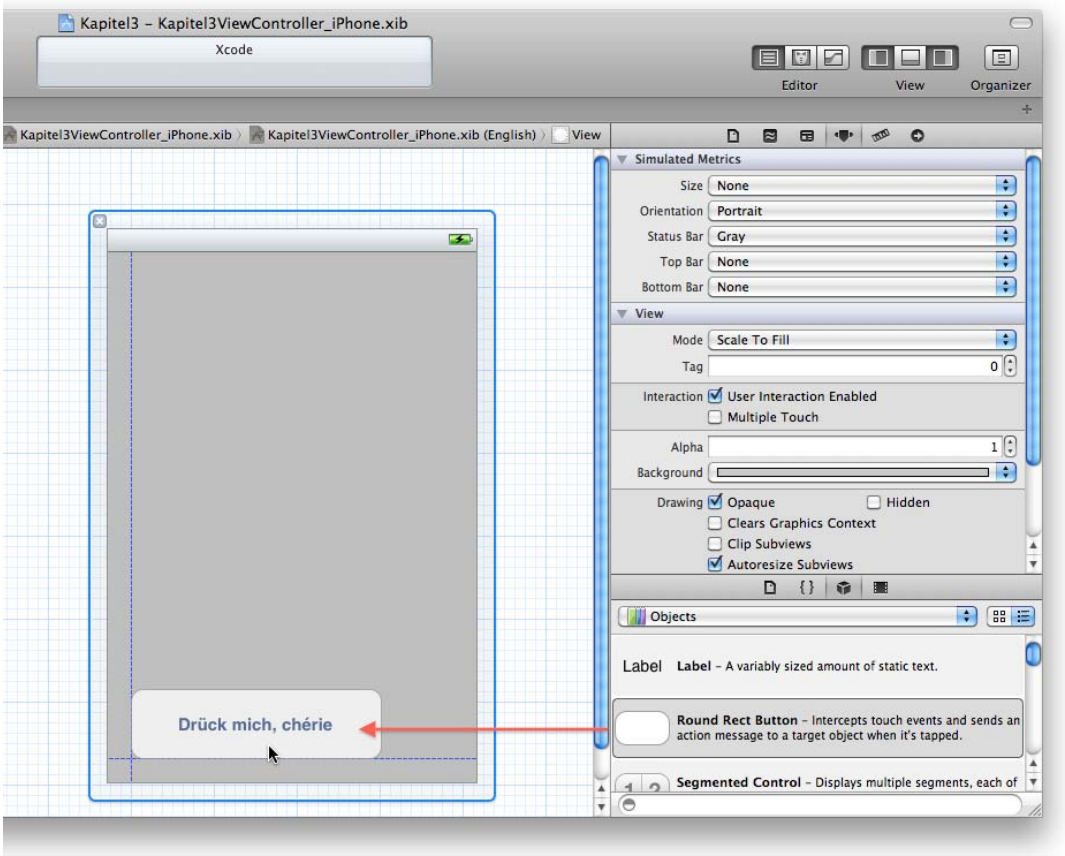

Abbildung 3.7: Per Drag&Drop ziehen Sie die Steuerelemente in das View und richten sie an den Hilfslinien aus.

Fügen Sie jetzt noch eine weitere Schaltfläche und ein Label-Element hinzu. Wenn Sie deren Größe ändern, sehen Sie in der Mitte eine gestrichelte vertikale Linie, die Ihnen anzeigt, dass das aktuelle Element nun an dieser Achse zentriert ist.

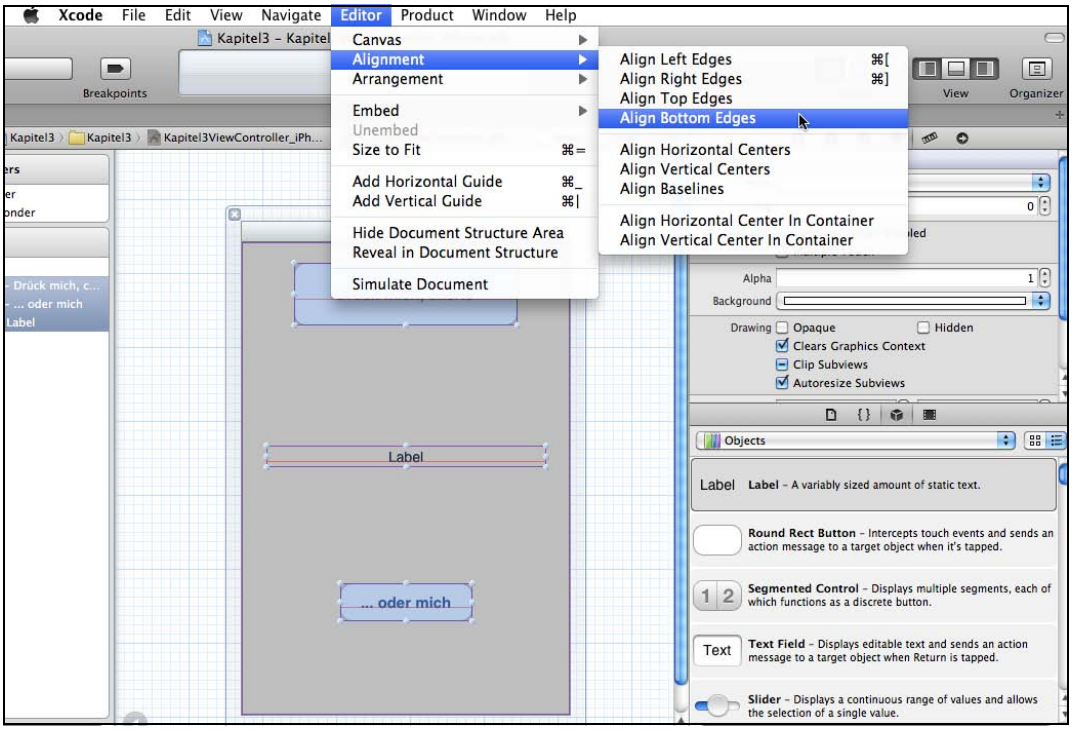

Abbildung 3.8: Bei der Gestaltung der Oberfläche verhält sich der Interface Builder fast wie ein Vektorgrafikprogramm.

Nachdem Sie die drei Steuerelemente zentriert auf dem Bildschirm angeordnet und die beiden Schaltflächen mit DRÜCK MICH und ... ODER MICH beschriftet haben, setzen Sie noch ein Attribut des Label-Steuerelements. Der Text, den wir später per Code einfügen wollen, soll nämlich zentriert dargestellt werden. Markieren Sie das Label-Element, und klicken Sie im Inspector rechts neben LAYOUT über ALIGNMENT auf das mittlere Symbol – ZENTRIERTE DARSTELLUNG.

Hinweis

Sämtliche Dateien, die noch gespeichert werden müssen, erkennen Sie in der linken Spalte des Xcode-Fensters daran, dass sie grau hinterlegt sind. Sie können sie einzeln speichern, indem Sie sie einzeln markieren und mit ⌘ + S sichern. Alternativ speichert Xcode aber auch alle Dateien, wenn Sie das Projekt starten.

3.3 Steuerelemente definieren

Der nächste Schritt besteht darin, sämtliche benötigten Steuerelemente im Code zu definieren und danach jene Event-Handler anzulegen, die ausgeführt werden, wenn eine der Schaltflächen gedrückt wird. Das ist jetzt komplett anders, als VisualBasic- oder Delphi-Programmierer es gewohnt sind, denn in VisualBasic und Delphi sind sämtliche Steuerelemente als Objekte, ihre Events und sämtliche Eigenschaften bereits vordefiniert und können einfach verwendet werden. Bei Objective-C ist dies nicht der Fall. Das muss der Programmierer selbst erledigen und anschließend die Instanzenvariablen mit den entsprechenden Objekten verbinden.

Die erste Begrifflichkeit, die Sie verstehen müssen, weil sie immer wieder vorkommt, ist das *IBOutlet*. Genau übersetzt, ist ein Outlet ein Ventil. Das *IBOutlet* („IB" steht für „Interface Builder") ist sozusagen das Interface-Builder-Ventil, also eine Schnittstelle zwischen Xcode und Interface Builder. Im Code sind sämtliche Steuerelemente, die stellvertretend über eine Objekt- oder Instanzenvariable angesprochen werden, als *IBOutlet* deklariert.

Kehren Sie zu dem Codebereich von Xcode zurück. In der linken Spalte sehen Sie unter dem Projektnamen – in unserem Falle *Kapitel3* – zwei Dateien, die zu dem aktuellen Interface Builder-View gehören, das wir gerade bearbeitet haben. Diese Dateien heißen *Kapitel3ViewController.h* und *Kapitel3ViewController.m*. In der Header-Datei befinden sich sämtliche Instanzenvariablen, also all jene Elemente, auf die Sie zugreifen können, während das *View* ausgeführt wird. Wir benötigen in dieser Headerdatei eine Instanzenvariable, die stellvertretend für das Label steht, das wir zuvor im Interface Builder platziert haben – sozusagen das, was man bei VisualBasic oder Delphi als Steuerelement kennt und über seine Name-Eigenschaft anspricht. In Objective-C müssen Sie eine eigenständige (Zeiger-)Variable deklarieren, um auf die einzelnen Steuerelemente – beispielsweise auf den Text (die Caption) des Labels – zugreifen zu können. Öffnen Sie die Headerdatei mit einem Doppelklick.In ihr ist – neben dem Kommentar – bereits die Interface-Deklaration enthalten. Erweitern Sie diese Deklaration um die neue Instanzenvariable *myLabel*:

```
@interface Kapitel3ViewController : UIViewController {
   IBOutlet UILabel *myLabel;
}
```

`*myLabel` ist ein Zeiger (erkenntlich an dem `*`) auf ein Interface Builder-Element (was an der Auszeichnung *IBOutlet* zu erkennen ist) vom Typ *UILabel*. Noch weiß jedoch Objective-C nicht, was es mit dieser Variablen auf sich hat, geschweige denn, dass es sich um jenes Steuerelement handelt, das Sie eben auf dem Formular abgelegt haben. Es geht nun also darum, diese Verbindung zwischen dem Code und dem Interface Builder herzustellen. Markieren Sie also in der linken Spalte die Ressource *Kapitel3ViewController.xib*. Markieren Sie hier in der zweiten Spalte von links direkt unter Placeholders den ersten Eintrag File´s owner, also den Besitzer der View-Gestaltungsdatei. Im letzten Reiter des Inspector – den Connections –, den Sie über den Shortcut ⌘+⌥+6 aktivieren können, sehen Sie alle Verknüpfungen zwischen der *.xib*-Datei und den zugehörigen Quelltextdateien, insbesondere natürlich der Headerdatei. Ganz oben sehen Sie auch die eben deklarierte Instanzenvariable *myLabel*. Klicken Sie rechts

daneben auf den Kreis, und ziehen Sie die Maus bei gedrückter Maustaste über das Label im gestalteten Formular. Sobald dieses als markiert hervorgehoben dargestellt wird, können Sie die Maustaste loslassen. Im Inspector sehen Sie nun, dass das Label-Steuerelement und das *IBOutlet UILabel* aus der Headerdatei miteinander verknüpft sind. Wenn Sie also vom Code aus auf die Objektvariable *myLabel* zugreifen, ändern Sie das Aussehen und Erscheinen des Label-Steuerelements des *View*.

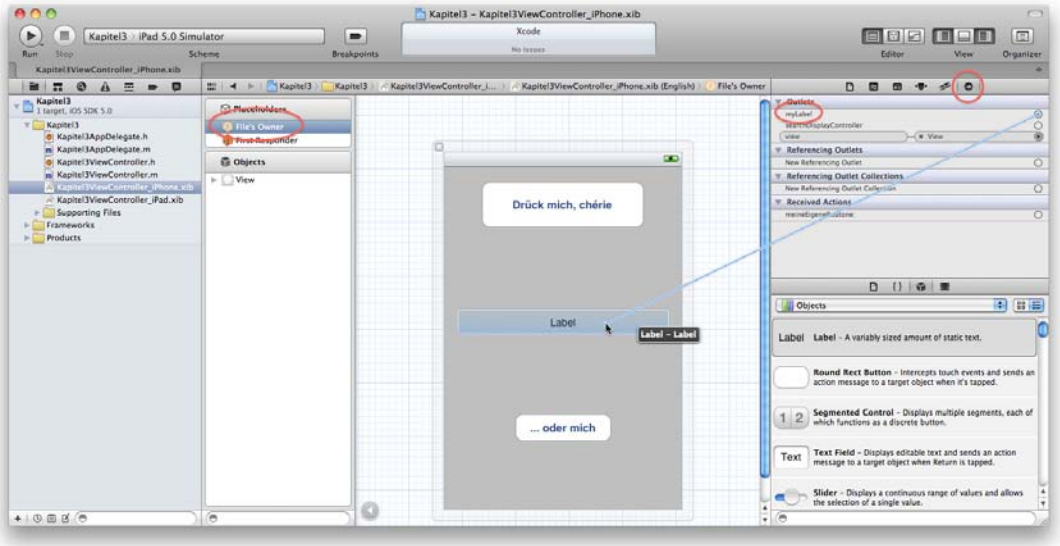

Abbildung 3.9: Per Drag&Drop wird die Instanzenvariable aus der Headerdatei mit den Steuerelementen verknüpft.

3.4 Auf Events reagieren

Als Nächstes deklarieren Sie in der Headerdatei des View (*Kapitel3ViewController.h*) zwei Methoden. Mit der Deklaration versprechen Sie sozusagen dem Compiler, dass an irgendeiner anderen Stelle – vornehmlich in der gleichnamigen Quellcodedatei – der Code für die genannten Methoden stehen wird. Zunächst aber brauchen wir nur die reine Deklaration. Die Methoden, die wir benötigen, sind jene, die ausgeführt werden, wenn der Benutzer auf eine der beiden Schaltflächen drückt. Nennen wir die beiden Methoden *pushEins* und *pushZwei*.

Eine Methode beginnt in der Regel mit einem -, gefolgt von der Art der Methode. Es gibt zwar auch noch die Möglichkeit, dass eine Methode mit einem + beginnt; innerhalb dieser Interface-Deklaration und weil wir auf Instanzenvariablen zugreifen, muss es jedoch hier das Minuszeichen sein. Erweitern Sie also in der Headerdatei die *UIViewController*-Deklaration um die beiden (fett hervorgehobenen) Methoden vom Typ *IBAction*. *IBAction* ist eine Direktive, die im ersten Durchlauf des Compilers, vom sogenannten Präprozessor, einfach in *void* übersetzt wird. Sie ist in der entsprechenden Headerdatei wie folgt definiert:

```
#define IBAction void
```

Warum Sie dann nicht gleich *void* schreiben können, werden Sie sich fragen. Der Grund ist, dass diese Behandlungsroutinen zur Entwicklungszeit im Interface Builder den entsprechenden Ereignissen der verschiedenen Schaltflächen, Labels, Listenfelder und was es sonst noch so gibt, zugeordnet werden sollen. Und der Interface Builder schaut in der gleichnamigen *.h*-Datei eben genau nach jenen Routinen, die mit *IBAction* ausgezeichnet sind.

```
@interface Kapitel3ViewController : UIViewController {
   IBOutlet UILabel *myLabel;
}
-(IBAction) pushEins;
-(IBAction) pushZwei;
@end
```

Das war's fürs Erste schon. Nun wechseln Sie wieder in den Interface Builder. Im Connection-Fenster des Inspectors (⌘+⌥+2) erscheinen die beiden Deklarationen nun als *Received Actions*. Ziehen Sie – wie auch zuvor bei den Eigenschaften – den kleinen Kreis rechts neben *pushEins* bei gedrückter Maustaste auf die erste Schaltfläche. Sobald Sie die Maustaste loslassen, erscheint eine Auswahl mit allen Ereignissen, die für das jeweilige Steuerelement verfügbar sind. Bei Schaltflächen sind natürlich Touch Down und Touch Up Inside besonders interessant. Das erste Ereignis wird dann abgefeuert, wenn der Benutzer die Schaltfläche nur antippt. Das zweite hingegen nur, wenn er den Finger in der Schaltfläche auch wieder loslässt. Das hat den Vorteil, dass der Benutzer es sich noch anders überlegen kann und gegebenenfalls den Finger außerhalb vom Display hebt. Verbinden Sie auf diese Weise *pushEins* mit dem Ereignis *Touch Down* der ersten Schaltfläche und *pushZwei* mit dem Ereignis *Touch Up Inside* der zweiten Schaltfläche.

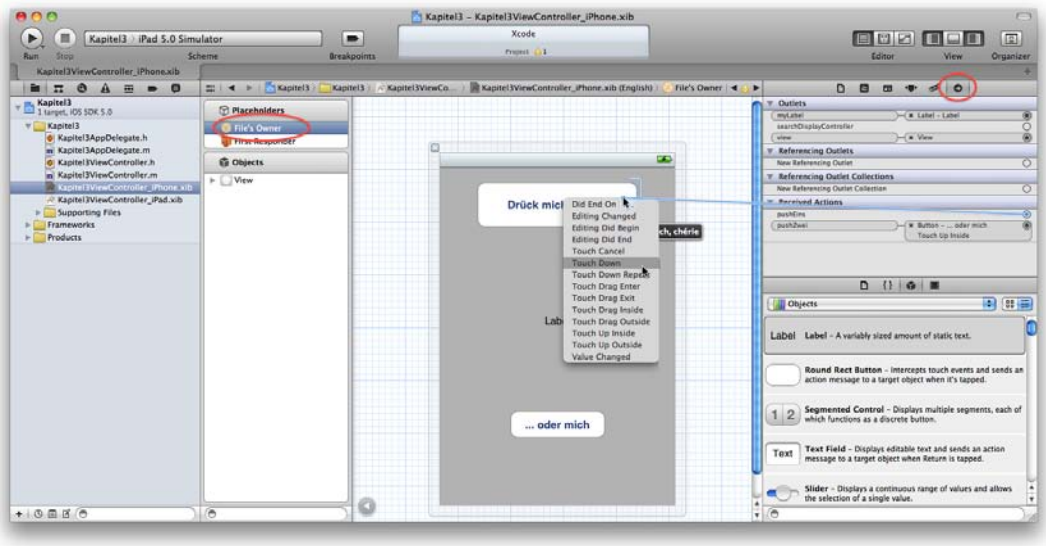

Abbildung 3.10: Auch die Methoden, die die verschiedenen Ereignisse der Schaltflächen behandeln, werden im Connections-Inspector mit den Steuerelementen verknüpft.

Die Verbindungen sind nun hergestellt, und es fehlt lediglich der Code, der ausgeführt werden soll, sobald der Benutzer die Schaltflächen berührt respektive im zweiten Fall loslässt. Dazu wechseln Sie wieder in den Editor von Xcode und öffnen die Datei *Kapitel3ViewController.m*. In dieser Datei sehen Sie bereits einige Ereignisbehandlungsroutinen. So zum Beispiel *viewDidUnload*, also die Stelle, die aufgerufen wird, wenn das View wieder vom Display entfernt wird. Zudem sind einige Behandlungsroutinen für wichtige Ereignisse bereits vordefiniert, aber auskommentiert. Fügen Sie nun nach der Zeile

```
@implementation Kapitel3ViewController
```

den Code für die beiden Behandlungsroutinen ein, die Sie zuvor in der Headerdatei deklariert und im Interface Builder den Schaltflächen zugewiesen haben:

```
-(IBAction) pushEins {
   [myLabel setText: @"Hallo Welt"];
}
```

> **Hinweis**
>
> Die Notation in der *.m*-Datei ist quasi dieselbe wie in der Headerdatei. Wenn Sie in der Headerdatei die Zeile mit der Deklaration kopieren und im *.m*-Modul einfügen, brauchen Sie nur das Semikolon zu markieren und `{ }` zu tippen; und schon ist die Funktionsdefinition wenigstens formal richtig.

In diesem Beispiel wird der Eigenschaft *myLabel* vom Typ *UILabel* über die *UILabel*-Methode *setText* die konstante Zeichenkette zugewiesen. Diese Zeichenkette ist vom Typ *NSString*. „NS" steht hierbei für *Next Step*. Dies war ein objektorientiertes Betriebssystem jener Firma *NeXT*, die Steve Jobs 1985 nach seinem „Fortgang" von Apple gegründet hatte. Später kaufte Apple Steve und dessen Firma wieder ein. *NeXTSTEP* (so die offizielle Schreibweise) ist die Basis für OS X und damit auch die Basis für Cocoa und Xcode. Zur Wahrung der Kompatibilität wurden die Klassennamen und Bezeichnungen der älteren Systeme beibehalten.

Die zweite Ereignisbehandlungsroutine macht im Grunde dasselbe, nur dass hier der Schritt über eine echte Variable vom Typ *NSString* gewählt wird:

```
-(IBAction) pushZwei {
   NSString *caption = @"Hallo Objective C";
   [myLabel setText: caption ];
}
```

Jetzt können Sie mit der Startschaltfläche das Projekt kompilieren, Interface Builder-Formular und Code vereinen und das fertige Programm im iPhone Simulator starten. Achten Sie darauf, dass im Codefenster links oben im Auswahlfeld auch der Simulator und nicht

das Device, also das reale iPhone-Gerät, ausgewählt ist. Solange Sie keine Signatur von Apple für Ihr iPhone besitzen, können Sie Programme nicht darauf ausführen lassen.

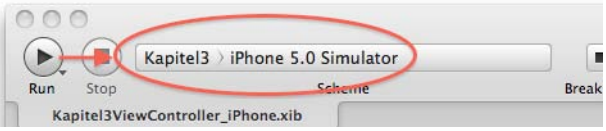

Abbildung 3.11: Beim Starten wird das aktuelle Projekt auf dem iDevice oder jenem Simulator gestartet, der links oben im Fenster ausgewählt ist.

Testen Sie nun einmal die beiden Schaltflächen. Wenn Sie auf die obere tippen, ändert sich schon beim Tippen selbst der Inhalt des Labels. Das kommt daher, weil das Ereignis, auf das reagiert wird, *Touch Down* ist. Bei der unteren hingegen – hier wurde als Ereignis *Touch Up Inside* gewählt – wird der Text erst dann in *Hallo Objective C* geändert, wenn der Benutzer den Finger wieder von der Schaltfläche hebt. Drückt er die Schaltfläche hingegen, schiebt dann den Finger auf dem Display aus dem Schaltflächenbereich und lässt ihn erst dort los, wird das Ereignis nicht ausgelöst. Das hat den Vorteil für den Anwender, dass er sich auch noch mal umentscheiden kann. Sie sollten also bei Schaltflächen zur Bestätigung oder zum Auslösen eines Vorgangs immer *Touch Up Inside* wählen. Statusbuttons hingegen reagieren sinnigerweise auf das einfache *Touch Down*-Event.

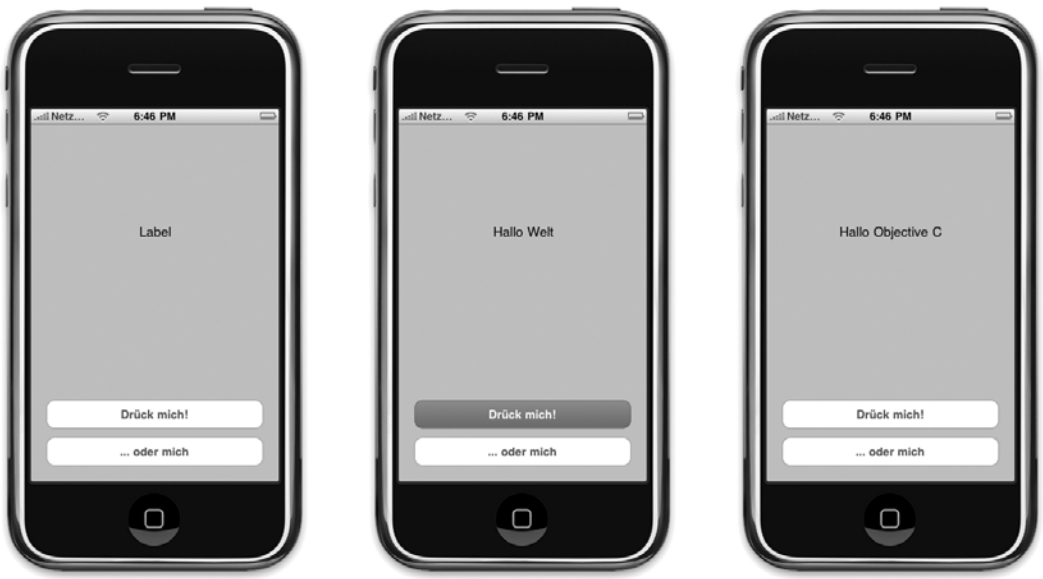

Abbildung 3.12: Während „Drück mich" bereits beim Antippen ausgeführt wird, muss man bei „oder mich" die Schaltfläche erst wieder loslassen.

3.5 IBAction und IBOutlet à la Xcode 4

„Mein Gott", werden Sie denken, „was ist das für ein Heidenaufwand, nur um ein paar Steuerelemente mit Leben zu erfüllen!" Im Grunde genommen gebe ich Ihnen recht. Sie haben recht, dass es eigentlich unnötig ist, so viel von Hand tippen zu müssen, wo das System doch eigentlich alle Informationen hat und Sie unterstützen könnte. Bis zur Version 3 von Xcode hat es das aber nicht getan. Aber wir sind jetzt zum Glück bei der Version 4 angelangt, und hier hat sich in puncto Vereinfachung der Deklaration und der Verbindung zwischen Interface Builder und dem Code-Bereich von Xcode einiges getan. Denn den ganzen Kram, den Sie eben noch mit Eintippen an den richtigen Stellen im Quellcode erledigen mussten (und den die Entwickler, die auf Xcode 3 angewiesen sind, immer noch auf diese Art erledigen müssen), wird nun durch einfaches Drag&Drop zustande gebracht.

Stellen wir uns der Einfachheit halber mal vor, dass wir dem Formular eine neue Schaltfläche hinzufügen, die beim Drauftippen irgendeinen Text als Button-Beschriftung anzeigen soll.

Als Erstes müssen Sie dazu in der Xcode 4-Oberfläche die Assistant Editor-Ansicht einschalten. Diese finden Sie in der oberen Leiste rechts, also dort, wo Sie auch die verschiedenen View-Einstellungen und den Organizer ein- und ausblenden können. Nur klicken Sie jetzt auf das mittlere Symbol über der Beschriftung Editor.

Abbildung 3.13:
Der neue Xcode-Automatismus funktioniert nur in der Assistant Editor-Ansicht.

Auf diese Weise wird eine horizontal unterteilte Darstellung angezeigt, in der – vorausgesetzt, eine Interface Builder-*.xib*-Datei ist markiert – oben das Formular und unten ein beliebiger Quelltext dargestellt ist. Welcher Quelltext angezeigt wird, können Sie bestimmen, indem Sie die entsprechende Datei in der Pfadangabe über den Quelltext wählen. In unserem Falle verwenden Sie die Headerdatei *Kapitel3ViewController.h*. Das Ganze müsste ungefähr so aussehen wie in Abbildung 3.14.

Legen Sie eine neue Schaltfläche auf das Formular. Klicken Sie diese mit der rechten Maustaste (oder mit der linken Maustaste bei gedrückter Ctrl-Taste) an, halten Sie die Maustaste gedrückt, und ziehen Sie eine Verbindungslinie in den Quelltext, und zwar genau in den Interface-Bereich vor die geschweifte Klammer zu }. Sobald Sie die Maustaste loslassen, öffnet sich ein Popup-Dialog, in dem Sie die Parameter für diese Schaltfläche festlegen können. Wählen Sie diese, wie in Abbildung 3.14 dargestellt. Sobald Sie das Popup mit der Taste Connect schließen, wird einerseits der *IBOutlet*-Quelltext für den Code-Platzhalter dieser Schaltfläche in die Headerdatei eingetragen. Andererseits wird aber auch gleichzeitig die Verbindung im Interface Builder hergestellt. Das bedeutet, die Arbeit ist getan.

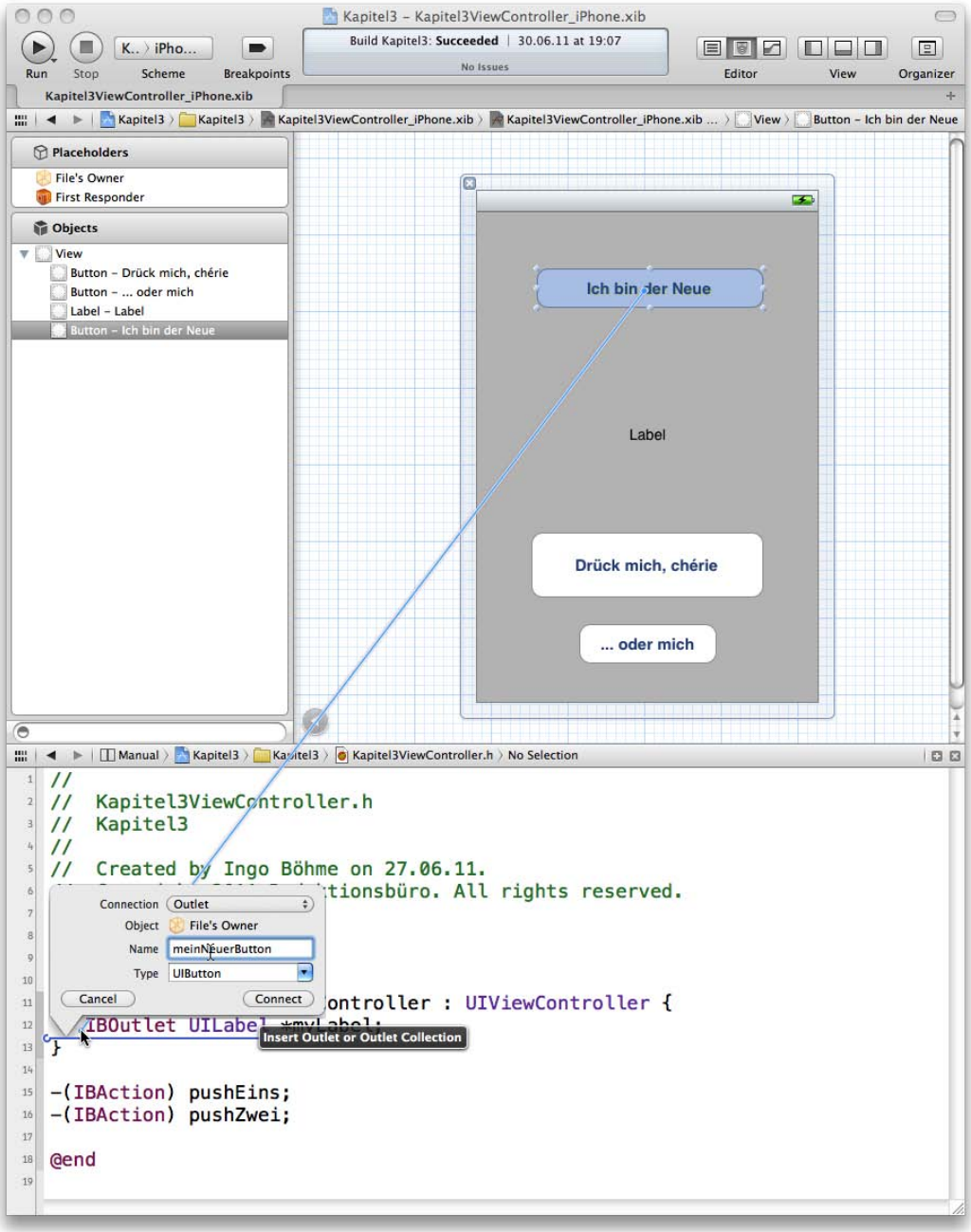

Abbildung 3.14: Wenn Sie ein Steuerelement mit der rechten Maustaste in den Interface-Bereich ziehen, wird eine IBOutlet-Variable angelegt.

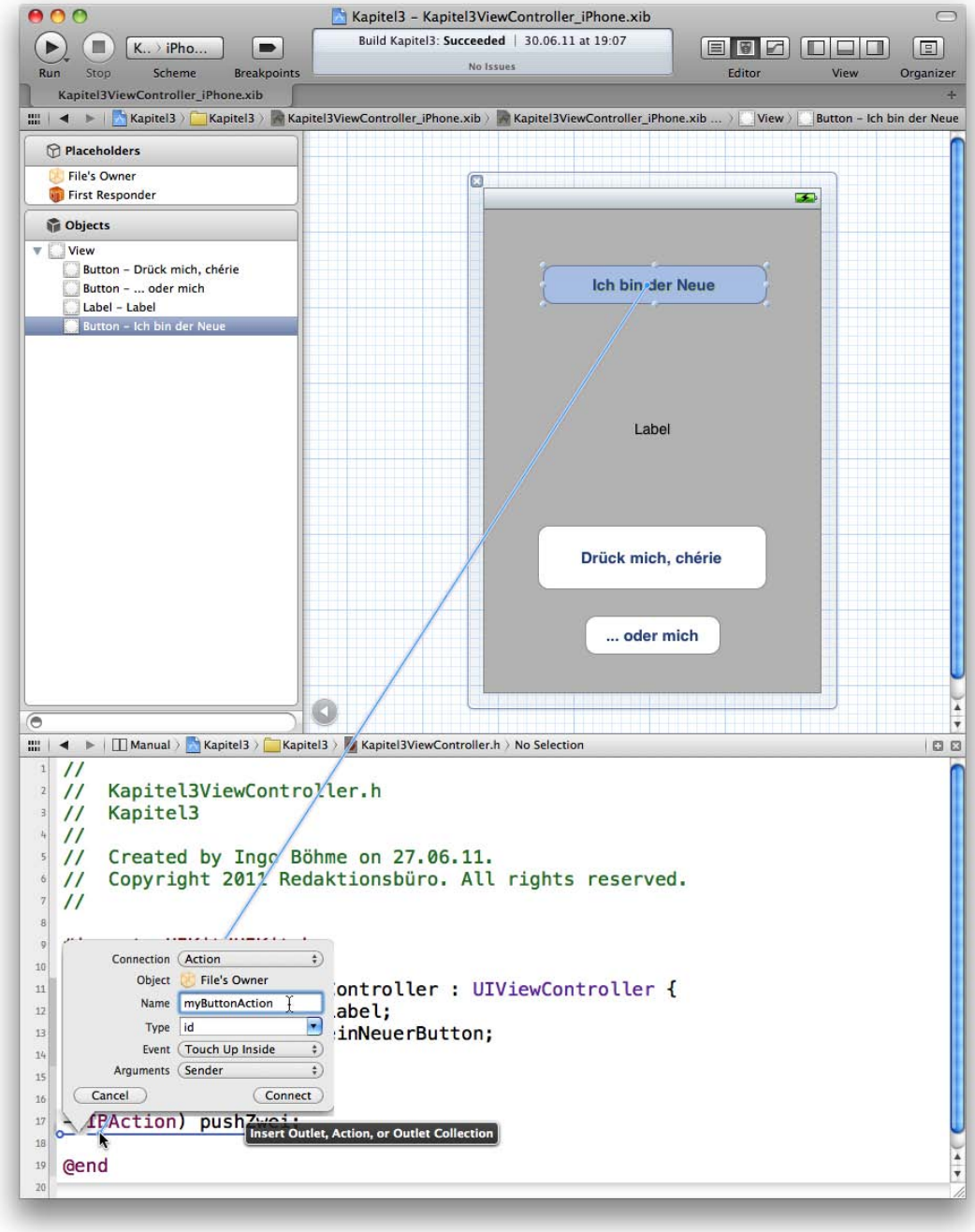

Abbildung 3.15: Ziehen Sie hingegen ein Steuerelement mit der rechten Maustaste direkt vor das @end, können Sie zudem eine IBAction-Methode für das Steuerelement erzeugen.

Da Sie neben der *IBOutlet*-Variablen *meinNeuerButton* auch noch eine *IBAction*-Methode brauchen, die aufgerufen wird, wenn die Schaltfläche gedrückt wird, gehen Sie noch einmal denselben Weg. Wenn Sie, wie in Abbildung 3.15 dargestellt, eine Verbindung hinter die geschlossene geschweifte Klammer ziehen, können Sie wie zuvor beim *IBOutlet* die Methodendeklaration einfügen lassen. Wie auch zuvor wird sowohl die Deklaration der Methode als auch die Verknüpfung mit der Schaltfläche automatisch hergestellt. Aber nicht nur diese Änderung in der Headerdatei gehört zum Automatismus. Auch in der dazugehörigen Moduldatei *Kapitel3ViewController.m* ist der Coderahmen bereits enthalten. Alles, was für Sie noch zu tun bleibt, ist diesen bereits vorgefertigten Rahmen mit Leben zu füllen, also beispielsweise mit dieser Codezeile:

```
[meinNeuerButton setTitle:@"Aldr, bin isch button" forState:UIControlStateNormal];
```

Hier nur kurz zur Erklärung: Ein Button besitzt zahlreiche Darstellungsarten – eine, wenn die Taste gedrückt ist; eine andere, wenn die Taste deaktiviert ist. Was wir jedoch machen wollen, ist im normalen Zustand der Taste die entsprechende Textzeile anzuzeigen. Dazu existiert die Methode *UIButton:setTitle:forState*. Dies ist eine Methode, die zwei Parameter benötigt. Der erste ist der anzuzeigende Text und der zweite der Status der Schaltfläche. Für diesen Status gibt es unterschiedliche Konstanten. Die Konstante, die den normalen Status beschreibt, heißt *UIControlStateNormal*. Der Methodenaufruf bedeutet also: „Weise der Schaltfläche den angegebenen Text für den normalen Status der Schaltfläche zu." Vielleicht versuchen Sie einmal, einen anderen Text für den Schaltflächenzustand *UIControlStateHighlighted* einzugeben. Schauen Sie einmal, was passiert ...

3.6 Eigene Methoden definieren

In unserem einfachen Beispiel haben wir direkt in die Ereignisbehandlungsroutine den Code geschrieben, der bei einem Klick ausgeführt werden soll. Dieser ist bei beiden Schaltflächen nahezu gleich. Und es wird lediglich der Text des Label-Steuerelements ausgetauscht. Wenn aber mehrere Aktionen ausgeführt werden, macht es Sinn, eine eigene Methode zu erzeugen, die sämtliche Anweisungen bündelt. Dann braucht man bei beiden Schaltflächen-Aktionen nur noch diese Methode aufzurufen und mit Parametern zu individualisieren.

Beginnen wir damit, das Programm so umzuschreiben, dass wir die Veränderungen über eine eigene Methode vornehmen. Der erste Schritt ist immer, in der Headerdatei die grundlegende Definition vorzunehmen, damit der Compiler Bescheid weiß, was auf ihn zukommt und womit er – im wahrsten Sinne des Wortes – rechnen muss.

Öffnen Sie die Headerdatei des View – *Kapitel3ViewController.h*. Direkt nach den beiden *IBAction*-Deklarationen erzeugen Sie eine neue Funktion, mit der zunächst der Text des Labels

verändert wird. Da der Rückgabewert nicht benutzt wird, wählen Sie den Datentyp *void*. Als Parameter soll der Text übergeben werden, der in dem Label angezeigt werden soll.

```
-(void) modifyLabel: (NSString*)caption;
```

Die Definition dieser Funktion nehmen Sie in der *.m*-Datei vor:

```
-(void) modifyLabel: (NSString*)caption {
   [myLabel setText: caption];
}
```

Statt des direkten Methodenaufrufs `setText` der *UILabel*-Instanz `myLabel`, können Sie jetzt die gerade fertiggestellte Methode `modifyLabel` verwenden, also für den ersten Button:

```
[self modifyLabel: @"Hallo Welt"];
```

Und für den zweiten:

```
[self modifyLabel: @"Hallo Objective C"];
```

Das Keyword `self` steht dabei für das View-Objekt.

Methoden ohne Parameter

Zuweilen gibt es natürlich auch Methoden, die keine Parameter besitzen. Diese werden in der Headerdatei einfach mit ihrem Namen notiert:

```
-(void) hugo;
```

Die Definition in der Quellcodedatei ist dann entsprechend:

```
-(void) hugo {
   [myLabel setText: @"Geht auch ohne Parameter"];
}
```

Auch der Aufruf entspricht dem mit einem Parameter, lediglich der Teil ab dem : fällt weg:

```
[self hugo];
```

Damit die Verwendung einer separaten Methode auch einen Sinn ergibt, soll neben dem Text auch gleich noch die Farbe des Labels verändert werden. Und eben diese Farbe soll als zweiter Parameter übergeben werden.

3.7 Methoden-Recherche in der Hilfe

Anstatt Ihnen jetzt zu sagen, welche Eigenschaft des Labels für die Textfarbe zuständig ist und wie deren Datentyp aussieht, gehen wir einen Schritt weiter in die Zukunft. Nämlich zu dem Zeitpunkt, wenn Sie es eben auch nicht wissen und es nicht gerade Thema dieses Buches ist. Dann nämlich ist die Zeit gekommen, um sich mit der Dokumentation auseinanderzusetzen.

Als Erstes müssen Sie die Dokumentation starten. Das geschieht über den ersten Punkt im HELP-Menü DEVELOPER DOCUMENTATION oder die Tastenkombination ⌘+⌥+⇧+?. Da Sie etwas über das Label-Objekt erfahren wollen, geben Sie im Suchfeld *UILabel* ein.

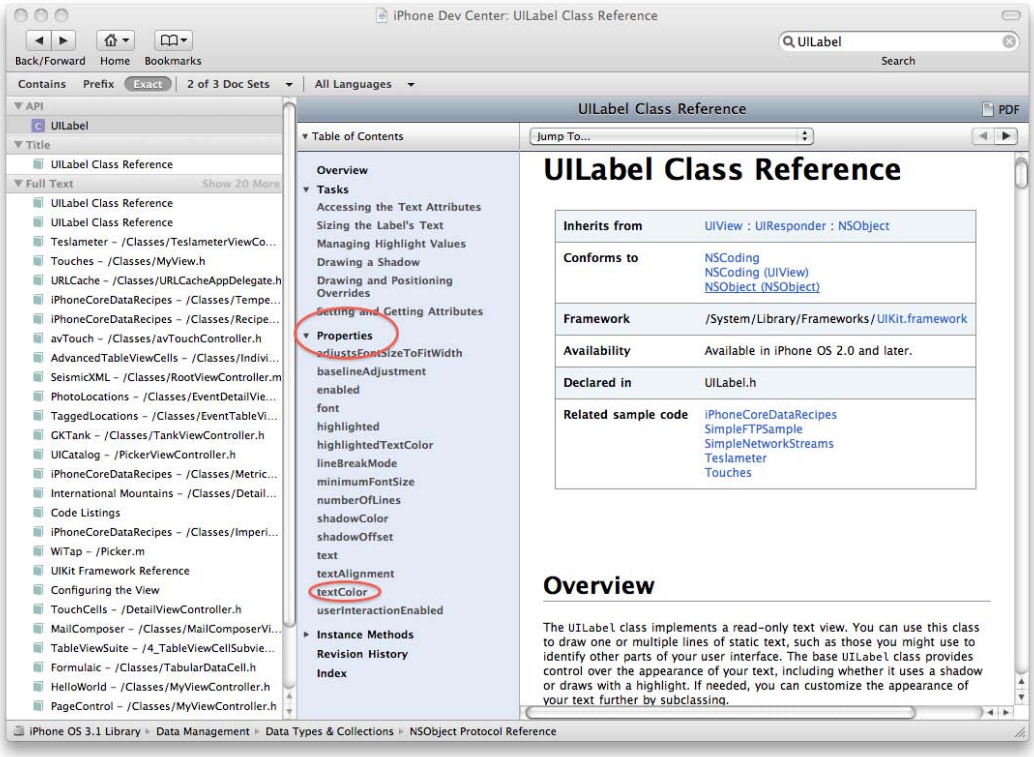

Abbildung 3.16: Mit der Hilfe schlagen Sie die Fähigkeiten, Eigenschaften und Methoden der einzelnen Objekte nach.

Da wir die Textfarbe verändern wollen, liegt es nahe, dass es irgendwo eine Eigenschaft – eine Property – gibt, die dafür zuständig ist. Also erweitern Sie in dem hellblau hinterlegten Bereich den Abschnitt PROPERTIES. Lesen Sie diese einmal durch. Man muss nicht gerade ein Hellseher sein, um darauf zu kommen, dass *textColor* die Eigenschaft der Wahl ist. Klicken Sie darauf, scrollt das Dokument zur richtigen Stelle.

Hier erfahren Sie, dass diese Eigenschaft vom Typ *UIColor* ist. Auch diesen Datentyp können Sie wieder anklicken und somit weitere Informationen über die Zusammenhänge herausfinden. Oder Sie gehen den einfachen Weg. Denn weiter unten sehen Sie den Text *Related Sample Code*, also Beispielcode, den Apple bereits mit dem SDK zur Verfügung stellt. Auch dieser Quellcode ist mit einem Link verknüpft, sodass Sie lediglich eine dieser fünf Varianten auswählen müssen.

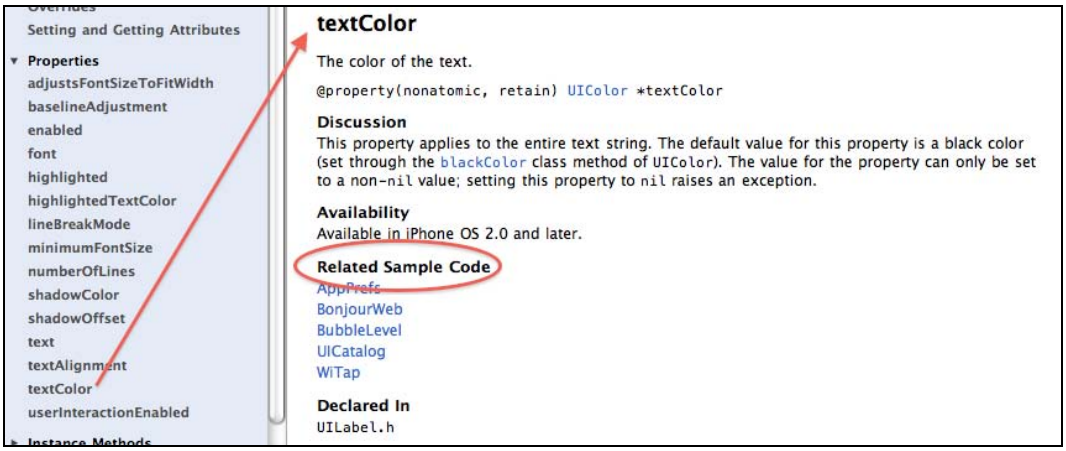

Abbildung 3.17: In der Hilfe sind die Verweise in die Beispielcode-Projekte verlinkt.

Klicken Sie einfach auf das erste Beispiel, *AppPrefs*. Welches Sie wählen, ist ja eigentlich egal, es interessiert uns ja nur, wie die Eigenschaft *textColor* im Code belegt wird. Nun öffnet die Hilfe nicht etwa Xcode mit dem Projekt, sondern eine Übersichtsseite mit einem Auswahlfeld, in dem sich sämtliche textorientierten Dateien des Projekts befinden. Klappen Sie dieses einmal aus. Hier sehen Sie nur drei Dateien, die in Frage kommen: *main.m, AppDelegate.m* und *MyViewController.m*. Denn das sind die einzigen Quelltextdateien, in denen auch Objective-C-Quellcode steht. Wie auch in unserem Beispiel, ist der *ViewController* der Bereich, in dem sich die Hauptaufgabe der Applikation befindet. Wählen Sie daher in dem Listenfeld den Eintrag MYVIEWCONTROLLER.M.

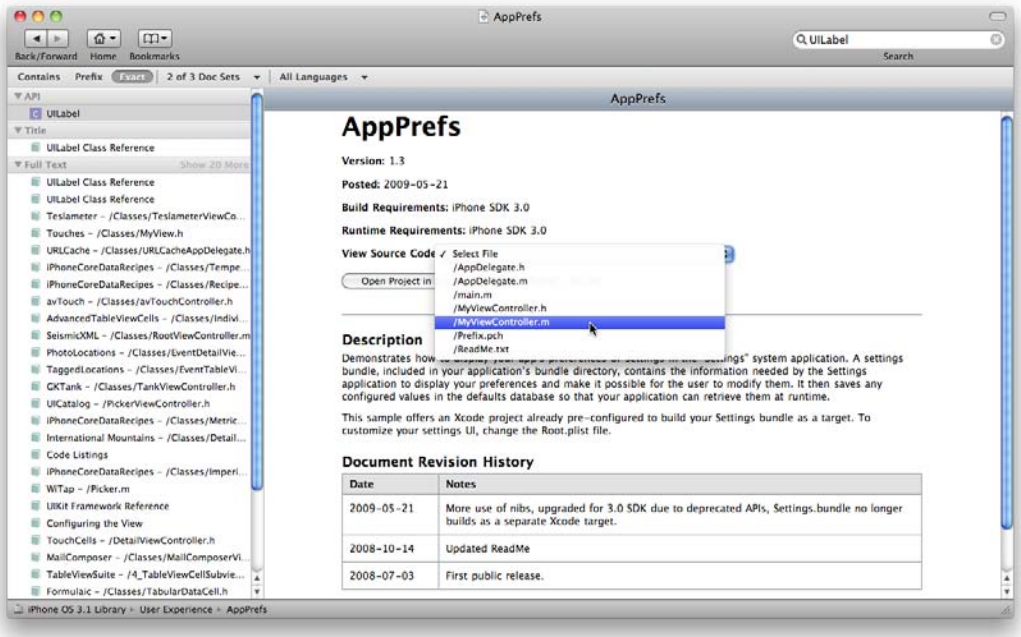

Abbildung 3.18: Wählen Sie im Listenfeld den Quelltext, wird er in der Hilfe angezeigt.

Je nach Menge des Quellcodes kann es ziemlich mühsam sein, den Teil zu finden, den man eigentlich sucht. Mit ⌘+F öffnen Sie ein Suchfeld über dem Quelltext. Wenn Sie dort die Namen der gesuchten Eigenschaft *textColor* eintragen, springt der Quelltext genau zur richtigen Stelle innerhalb des Beispieldokuments. Und dort sehen Sie ein Konstrukt in der Form:

```
switch ([appDelegate textColor])
    {
        case blue:
            cell.textLabel.textColor = [UIColor blueColor];
            break;
            // ... usw ...
```

Das bedeutet also, dass die Textfarbe eines Labels in Blau geändert wird, wenn die Zuweisung

```
LABELELEMT.textColor = [UIColor blueColor];
```

vorgenommen wird. Auf der rechten Seite der Zuweisung steht eine Methodenfunktion des Objekts *UIColor*, nämlich dessen Methode *blueColor*. Um zu erfahren, welche anderen Farben bereits vordefiniert sind, geben Sie in dem Suchfenster der Hilfe *UIColor* ein. Dort erfahren Sie wiederum im Abschnitt *Class Methods*, dass es Varianten gibt, wie etwa *purpleColor*, *redColor* oder auch *yellowColor*.

3.8 Die Label-Farbe ändern

Nun wissen Sie, dass die Textfarbe über die Eigenschaft *textColor* gesetzt wird und dass diese Eigenschaft vom Typ *UIColor* ist. Erweitern Sie jetzt in der *.m*-Datei Ihres Projekts den Code der Methode *modifyLabel* um die Zeile

```
myLabel.textColor = [UIColor redColor];
```

und starten Sie das Programm im Simulator mit Build and Run. Sobald Sie nun auf die Schaltfläche tippen, wird nicht nur der Text, sondern auch die Farbe des Labels geändert.

3.9 Eine Methode mit zwei Parametern

Soll – je nach Schaltfläche – eine andere Farbe erscheinen, können Sie dies beispielsweise lösen, indem Sie die Methode *modifyLabel* mit zwei Parametern ausstatten. Der erste ist und bleibt der Text, und der zweite soll die Farbe sein. Und nun kommt etwas, was auf den ersten Blick völlig verwirrt. In anderen Programmiersprachen – und das geht weit über PHP, VisualBasic, Delphi oder auch C++ hinaus – werden Parameter an Methoden einfach kommasepariert übergeben. Wer sich vielleicht noch an COBOL erinnert (ja, ich war einer der Leidtragenden, die diese Sprache lernen mussten), der wird vielleicht noch die langen, fast an natürliche Sprache erinnernden Befehle vor Augen haben. Als Assembler-Programmierer habe ich mir damals gedacht: „Welcher zehnfingertippende Idiot hat sich so etwas ausgedacht? Da schreib ich mir mit meinem Zwei-Finger-System die Kuppen wund." Das Argument der COBOL-Fans war immer, dass der Quelltext später besser lesbar und damit leichter zu bearbeiten sei. Na ja, Cobol gibt´s quasi nicht mehr. Mich schon noch ...

„Warum dieser Griff in die Historie?", werden Sie sich fragen. Nun, Objective-C verwendet – auch wenn es ganz ohne Zweifel ansonsten überhaupt nichts mit der Sprache COBOL zu tun hat – beim Aufruf von Methoden ein ähnlich langatmiges und wortreiches Verfahren. Anstatt die einzelnen Parameter einfach mit einem Komma zu trennen, wird bei Objective-C-Methoden ab dem zweiten Parameter eine Beschreibung vorangestellt, auf die ein Doppelpunkt folgt. Diese Beschreibung muss aus einem Wort bestehen. Daher bietet sich auch hier das *camelCasing* an, weil man damit eine echte Beschreibung angeben kann. Da bei uns im Beispiel der zweite Parameter die Farbe sein soll, in der das Label erscheint, ist als Beschreibung beispielsweise *inDerFarbe:* geeignet. Ändern Sie also in der Headerdatei die Zeile der Methodendeklaration für *modifyLabel* wie folgt ab:

```
-(void) modifyLabel: (NSString*)caption  inDerFarbe: (UIColor*)farbe;
```

Ganz analog dazu sieht die Zeile in der eigentlichen Quellcodedatei *Kapitel3View.m* aus:

```
-(void) modifyLabel: (NSString*)caption inDerFarbe:(UIColor*)farbe {
```

Der einzige Unterschied ist wieder, dass statt des ; in der Headerdatei im Quellcode die geöffnete geschweifte Klammer steht.

Innerhalb der Funktion brauchen Sie nur den Farbwert zuzuweisen, also der Eigenschaft *textColor* des Steuerelements *myLabel* den Wert des Parameters *farbe*:

```
myLabel.textColor = farbe;
```

Damit nun auch die Farbe verändert wird, passen Sie die beiden *IBAction*-Routinen für die Schaltflächen-Aktionen an den veränderten Methodenaufruf von *modifyLabel* an, also

```
[self modifyLabel: @"Hallo Welt"  inDerFarbe: [UIColor yellowColor]];
```

für die erste Schaltfläche und

```
[self modifyLabel: @"Hallo Objective C"  inDerFarbe: [UIColor blueColor ]];
```

für die zweite. `[UIColor blueColor]` bedeutet dabei, dass die *UIColor*-Methode *blueColor* ausgeführt und deren Rückgabe als zweiter Parameter für unsere Objektfunktion *modifyLabel* verwendet wird.

Und hier sehen Sie auch den Vorteil dieser Notation von Objective-C: Die Methodenaufrufe sind sprechend. Auch wenn Sie dieses Kapitel und dieses Beispiel lange vergessen haben, werden Sie, wenn Sie diese Zeilen sehen, sicher sofort wissen, was dort vor sich geht.

Starten Sie nun noch einmal das Programm. Und tatsächlich ändern sich sowohl der Text als auch die Farbe.

3.10 Rückblick und Ausblick

Was machen Sie noch hier? Eigentlich können Sie jetzt anfangen zu programmieren. Durchstöbern Sie die Dokumentation! Werfen Sie ein paar Steuerelemente auf ein View, und los geht´s. Probieren Sie aus! Denn das grundsätzliche Rüstzeug haben Sie jetzt. Viel mehr braucht es wirklich nicht. Das einzig Schwierige, was es jetzt noch gibt, ist, dass die Bibliotheken, also die Frameworks des iOS SDK enorm groß sind. Der Überblick ist das, was Sie jetzt noch nicht haben. Wie genau Sie es anstellen, dass Ihre Applikation eine Webseite anzeigt. Was passiert, wenn Sie das Gerät drehen usw. Aber keine Bange: Die wichtigsten Aktionen, die Sie in 90 % aller iPhone-Applikationen vorfinden, werden wir in den kommenden Kapiteln behandeln.

Copyright