

# **Softwareentwicklung für iOS mit Objective-C und Xcode**

**Skript**

Nils Fischer

Universität Heidelberg - Wintersemester 2013/14

# Inhaltsverzeichnis

<b>1 Organisatorisches</b>	<b>5</b>
1.1 Über meine Person . . . . .	5
1.2 Über diesen Kurs . . . . .	5
1.3 Hardware und Software . . . . .	6
1.4 Über dieses Skript . . . . .	6
1.5 Dokumentationen und Referenzen . . . . .	7
<b>2 Xcode</b>	<b>8</b>
2.1 Workspaces & Projekte . . . . .	8
2.1.1 Neue Projekte anlegen . . . . .	8
2.1.2 Targets und Products . . . . .	10
2.1.3 Projekt- und Target-Konfiguration . . . . .	11
2.2 Benutzerinterface . . . . .	12
2.2.1 Darstellungsmodi . . . . .	13
2.2.2 Build & Run . . . . .	14
2.2.3 Navigator . . . . .	14
2.2.4 Editor . . . . .	15
2.2.5 Inspektor . . . . .	18
2.2.6 Debug-Bereich & Konsole . . . . .	19
2.3 Interface Builder . . . . .	20
2.3.1 XIBs & Storyboards . . . . .	21
2.3.2 Inspektor im IB-Modus . . . . .	22
2.3.3 IBOutlets & IBActions . . . . .	22
2.4 Dokumentation . . . . .	24
2.5 Testen auf iOS Geräten . . . . .	24
2.5.1 Der Provisioning Prozess . . . . .	26
2.5.2 Provisioning mit Xcode . . . . .	27
<b>3 Objective-C</b>	<b>31</b>
3.1 Grundlagen der Programmierung . . . . .	31
3.1.1 Primitive Datentypen . . . . .	32
3.1.2 Kommentare . . . . .	32
3.1.3 Output . . . . .	32
3.1.4 Einfache Operationen . . . . .	33
3.1.5 Abfragen . . . . .	33
3.1.6 Schleifen . . . . .	34

## Inhaltsverzeichnis

3.1.7	Strings . . . . .	35
3.1.8	String Formatierung . . . . .	35
3.2	Grundlagen der objektorientierten Programmierung . . . . .	36
3.2.1	Klassen & Objekte . . . . .	36
3.2.2	Interface & Implementierung . . . . .	37
3.2.3	Attribute . . . . .	37
3.2.4	Methoden . . . . .	38
3.2.5	Instanz- und Klassenmethoden . . . . .	39
3.2.6	Polymorphie . . . . .	39
3.2.7	Getter und Setter Methoden . . . . .	40
3.2.8	Instanzierung von Objekten . . . . .	42
3.2.9	Verfügbarkeit von Klassen . . . . .	42
3.3	Symbolnamen & Konventionen . . . . .	43
3.4	Einige wichtige Klassen . . . . .	44
<b>4</b>	<b>iOS App Architektur</b>	<b>46</b>
4.1	iOS App Lifecycle . . . . .	46
4.1.1	App States . . . . .	46
4.1.2	Startprozess einer iOS App . . . . .	47
4.2	Das Model-View-Controller Konzept . . . . .	50
4.3	View Hierarchie . . . . .	51
4.3.1	Frame und CGRect . . . . .	51
4.3.2	UIView Objekte . . . . .	53
4.4	Auto Layout . . . . .	53
4.4.1	Constraints . . . . .	54
4.4.2	Intrinsic Content Size . . . . .	54
4.4.3	Auto Layout im Interface Builder . . . . .	55
4.4.4	Auto Layout im Code . . . . .	57
4.5	View Controller Hierarchie . . . . .	58
4.5.1	View Controller Lifecycle . . . . .	60
4.5.2	Präsentation von View Controllern . . . . .	61
4.5.3	Container View Controller in UIKit . . . . .	61
4.5.4	View Controller in Storyboards . . . . .	62
4.6	Das Delegate Konzept . . . . .	64
4.7	Table Views & Table View Controller . . . . .	67
4.7.1	Statische und dynamische Table Views . . . . .	68
4.7.2	Datasource, Delegate und Table View Controller . . . . .	69
4.8	Data Persistance . . . . .	71
4.8.1	User Defaults . . . . .	72
4.8.2	Core Data . . . . .	73
4.8.3	State Preservation . . . . .	74
4.9	Notifications . . . . .	74

## *Inhaltsverzeichnis*

<b>5 Entwicklungsprozess von iOS Apps</b>	<b>76</b>
5.1 Versionskontrolle mit Git . . . . .	76
5.1.1 Grundlagen der Kommandozeilensyntax . . . . .	76
5.1.2 Git Repository & Commits . . . . .	76
5.1.3 Branches . . . . .	77
5.1.4 Zusammenarbeit mit Git & GitHub . . . . .	78
5.1.5 Git in Xcode . . . . .	79
5.1.6 Gitignore . . . . .	79
5.1.7 Dokumentation . . . . .	81
5.2 Design & Konzeption von iOS Apps . . . . .	82
5.2.1 App Statement . . . . .	82
5.2.2 Mockups . . . . .	83
5.2.3 Human Interface Guidelines . . . . .	83
5.3 Veröffentlichung im iOS App Store . . . . .	84
5.3.1 Entwickleraccount . . . . .	84
5.3.2 Unternehmensanmeldung . . . . .	84
5.3.3 iTunesConnect . . . . .	85
5.3.4 App Review . . . . .	86
5.3.5 Geschäftsmodelle . . . . .	86
5.3.6 Marketing . . . . .	89
5.3.7 Statistik . . . . .	90

# 1 Organisatorisches

## 1.1 Über meine Person

Mein Name ist Nils Fischer und ich bin Physikstudent im 3. Semester an der Uni Heidelberg. Mit der Softwareentwicklung für iOS Geräte beschäftige ich mich nun schon seit einigen Jahren und bin mit drei eigenen und fünf Apps für Kunden im App Store vertreten.

Diesen Kurs halte ich dieses Semester zum ersten Mal und freue mich immer über Feedback und Vorschläge von euch.

Ihr könnt mich jederzeit über meine Email [1] erreichen.

Außerdem steht für organisatorische Fragen Prof. Dr. Peter Fischer [2] als Ansprechpartner zur Verfügung.

## 1.2 Über diesen Kurs

Der Kurs findet im **Mac-Pool (Medienzentrum, INF 293, Raum 214)** des **URZ** statt.

**Jeden Montag um 16h ct.** während des Semesters wird es zunächst eine Vorlesungseinheit über ein neues Thema geben und ein zugehöriges Übungsblatt verteilt. Im direkt anschließenden Übungsteil besprechen wir dann das Übungsblatt der vorigen Woche, dessen Lösungen vorzugsweise von einem von euch vorgestellt werden. Außerdem kann das neue Übungsblatt bearbeitet werden und es gibt Raum für Fragen zur Vorlesung.

Um langfristig mitzukommen ist es am Wichtigsten, die wöchentlichen Übungen selbst zu bearbeiten und nicht nur zuzuhören und Code von anderen zu kopieren. Für die regelmäßige Anwesenheit und Mitarbeit in den Übungen werden **2 LP** vergeben. Es wird keine Klausur oder Note geben.

---

<sup>1</sup>n.fischer@stud.uni-heidelberg.de

<sup>2</sup>peter.fischer@ziti.uni-heidelberg.de

## 1.3 Hardware und Software

Für die Teilnahme an diesem Kurs ist Zugang zu einem Intel-Mac mit Mac OS 10.7.4 Lion oder neuer (empfohlen Mac OS 10.8.4 Mountain Lion oder neuer) erforderlich. Im Mac-Pool stehen 8 Mac Pro's für diejenigen Teilnehmer zur Verfügung, die keinen eigenen Mac mitbringen. Allen anderen wird jedoch empfohlen, mit ihren eigenen Macs zu arbeiten.

Wir arbeiten fast ausschließlich mit **Xcode**, Apple's Integrierter Entwicklungsumgebung (IDE), in Version 4.2 oder neuer (empfohlen Version 5.0 oder neuer). Die neueste Version kann und sollte schon vor Beginn des Kurses im Mac App Store (kostenlos) heruntergeladen und installiert werden [<sup>3</sup>].

Die Installation von Xcode enthält dann das iOS SDK (Software Development Kit) entsprechend in Version 5.0 (bei Xcode Version 4.2) bis zu 7.0 (bei Xcode Version 5.0) oder neuer.

Screenshots und Referenzen auf Bedienelemente in Xcode werden in diesem Skript in englischer Sprache sein. Es bietet sich ohnehin an, die englische Sprachversion von Xcode zu installieren, da ein Großteil der online verfügbaren Dokumentationen und Tutorials auf englisch ist.

Um eigene Apps im Simulator zu testen, ist keine weitere Konfiguration nötig. Zum Testen auf eigenen iOS Geräten (iPhone / iPad / iPod Touch) ist ein Lizensierungsprozess erforderlich, den wir noch durchführen werden (*s. S. 24, Abschnitt 2.5*). Führt bitte ein Softwareupdate eurer Geräte auf die neueste Version durch.

## 1.4 Über dieses Skript

Dieses Skript wird im Verlauf des Semesters im Moodle [<sup>4</sup>] kapitelweise zur Verfügung gestellt. Parallel dazu wird es für die Apps, die wir im Rahmen des Kurses erstellen werden, ein weiteres Dokument geben. Dieser **App-Katalog** wird ebenfalls im Moodle zu finden sein und enthält außerdem die Übungsaufgaben, die wöchentlich zu bearbeiten sind.

Die Kursinhalte werden sich thematisch an der Struktur des Skriptes orientieren und ihr könnt es als Referenz bei der Lösung der Übungsaufgaben verwenden. Mit der dokumentübergreifenden Suche (⌘ + F) lässt sich gut nach Stichwörtern suchen.

Das Skript ist kein Tutorial, sondern ist sehr allgemein gehalten und erläutert die Grundlagen der Kursthemen. Im ergänzenden App-Katalog findet ihr hingegen Schritt-für-Schritt Anleitungen.

---

<sup>3</sup><https://itunes.apple.com/de/app/xcode/id497799835?mt=12>

<sup>4</sup><http://elearning2.uni-heidelberg.de/course/view.php?id=3359>

Die Inhalte basieren auf der Xcode Version 5.0 und dem iOS SDK 7.0. Die meisten Erläuterungen und Screenshots lassen sich ebenso auf frühere Versionen von Xcode und dem iOS SDK beziehen, doch einige neuere Funktionen sind der aktuellen Version vorbehalten.

## 1.5 Dokumentationen und Referenzen

Zusätzlich zur in Xcode integrierten und online verfügbaren Dokumentation (*s. S. 24, Abschnitt 2.4*) bietet Apple einige Ressourcen für iOS Developer an:

**iOS Dev Center** [5] ist Apple's Onlineplattform für iOS Entwickler. Hier ist auch das Member Center zur Accountverwaltung und das Provisioning Portal zur Verwaltung der Certificates und Provisioning Profiles (*s. S. 26, Abschnitt 2.5.1*) zu finden.

**iOS Human Interface Guidelines (HIG)** [6] ist ein Dokument, das jeder iOS Developer gelesen haben sollte. Die hier besprochenen Richtlinien bezüglich der Gestaltung von Benutzeroberflächen auf der iOS Plattform sind sehr aufschlussreich und haben sicherlich ihren Teil zum Erfolg der iOS Geräte beigetragen. Ein entsprechendes Dokument gibt es auch für Mac [7].

**iOS App Programming Guide** [8] stellt eine Übersicht über die Architektur von iOS Apps dar.

**WWDC Videos** [9] werden während der jährlichen Worldwide Developer Conference veröffentlicht. Apple Entwickler führen sehr anschaulich in neue, grundlegende und fortgeschrittene Technologien und Methoden ein und geben Best-Practices.

Hier darf natürlich auch die Community-basierte Q&A-Seite **Stack Overflow** [10] nicht unerwähnt bleiben, die bei Codefragen immer sehr hilfreich ist.

---

<sup>5</sup><https://developer.apple.com/devcenter/ios/>

<sup>6</sup><https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/>

<sup>7</sup><https://developer.apple.com/library/mac/documentation/userexperience/Conceptual/AppleHIGuidelines/>

<sup>8</sup>[https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphonesosprogrammingguide](https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphonesosprogrammingguide/)

<sup>9</sup><https://developer.apple.com/wwdc/videos/>

<sup>10</sup><http://stackoverflow.com>

# 2 Xcode

Xcode ist Apple's Integrierte Entwicklungsumgebung (IDE) und wird von allen Entwicklern verwendet, die native Apps für die iOS oder Mac Plattformen schreiben. Siehe auch 1.3 für Informationen zu den Hardware- und Softwarevoraussetzungen für diesen Kurs.

Wir werden lernen, Xcode's zahlreiche Funktionen zu nutzen, um nicht nur effizient Programmcode zu schreiben, sondern auch unsere Programmierprojekte zu verwalten, Apps auf eigenen iOS Geräten zu testen, Benutzeroberflächen zu gestalten, Datenstrukturen zu erstellen und unsere Apps schließlich zu veröffentlichen.

IDE's anderer Systeme bieten häufig ein ähnliches Funktionsspektrum und der Umgang mit diesen ist somit leicht übertragbar. Natürlich kann Programmcode auch mit einem beliebigen Texteditor geschrieben werden, doch IDE's wie Xcode erleichtern die Programmierung häufig erheblich. Allein schon die umfangreiche Autovervollständigung vereinfacht das Schreiben von Objective-C Code sehr und korrigiert Syntaxfehler.

## 2.1 Workspaces & Projekte

Jedes Programmierprojekt wird in Xcode in Form eines **Projekts** angelegt. Diese können in **Workspaces** zusammengefasst werden. Letztendlich wird immer ein Workspace angelegt, auch wenn er nur ein Projekt enthält.

### 2.1.1 Neue Projekte anlegen

Mit **File** **New** **Project...** oder **⌘ + ⌘ + N** erstellen wir ein neues Projekt und wählen im erscheinenden Dialogfenster (s. S. 9, Abb. 2.1) ein Template. Zum Ausprobieren der Grundlagen von Objective-C eignet sich das **OS X** **Application** **Command Line Tool**. In diesem Dialogfenster finden wir mit **Cocoa Application** auch das Template, um Mac OS X Apps zu schreiben. Für iOS Apps verwenden wir die Templates unter **iOS** **Application**. Es kann prinzipiell mit einem beliebigen Template begonnen werden, diese unterscheiden sich nur in bereits vorkonfigurierten Programmelementen. Häufig ist es hilfreich, eines der Templates zu verwenden, das der Struktur der App entspricht, die wir programmieren wollen. Andernfalls kann mit **Empty Application** eine komplett leere App erstellt werden.

Nach der Wahl des Templates werden weitere Optionen für das Projekt präsentiert (s. S. 10, Abb. 2.2).

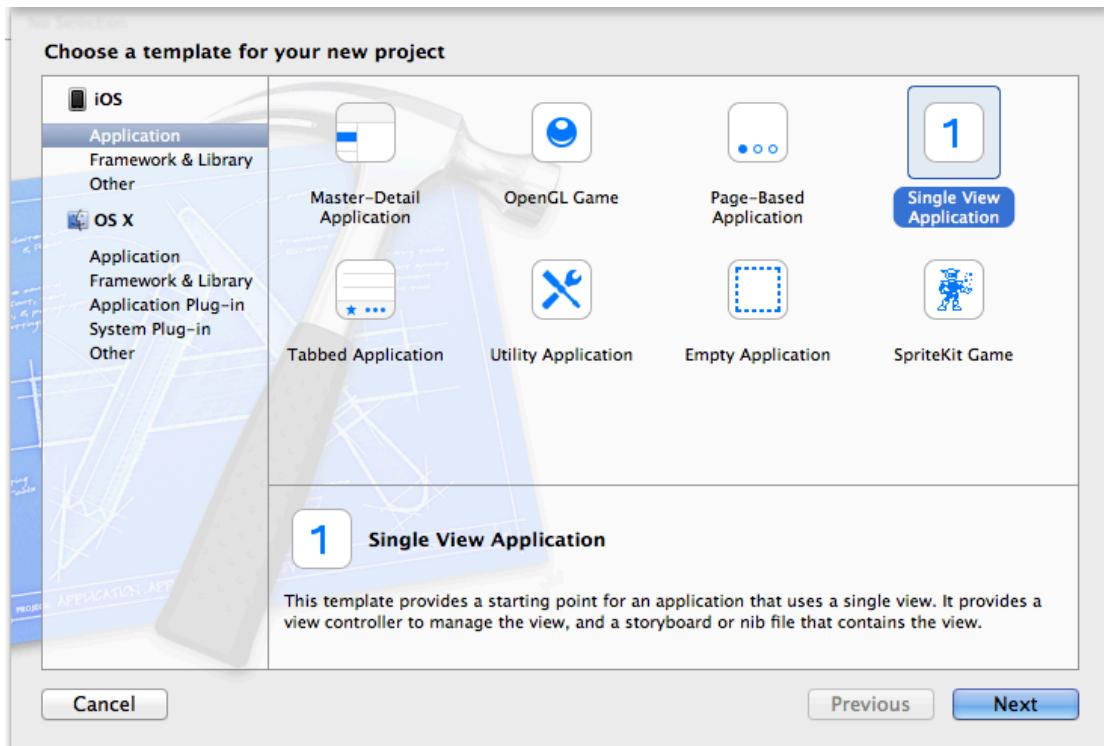


Abbildung 2.1: Ein neues Xcode-Projekt anlegen

**Product Name** identifiziert die resultierende App im Projekt. Es können später in einem Projekt weitere Produkte hinzugefügt werden (*s. S. 10, Abschnitt 2.1.2*). Diesen müssen unterschiedliche Product Names zugewiesen werden. Als Product Name wird häufig ein kurzer Codename des Projekts gewählt.

**Organization Name** wird in jeder erstellten Dateien hinterlegt.

**Company Identifier** identifiziert den Ersteller der Produkte. Nach Konvention wird hier eine sog. 'Reverse DNS' verwendet mit dem Schema 'com.yourcompany'. Verwendet in diesem Kurs bitte immer 'de.uni-hd.deinname' als Company Identifier.

**Bundle Identifier** setzt sich nach dem 'Reverse DNS' Schema aus Company Identifier und Product Name zusammen und hat dann die Form 'com.yourcompany.productname' bzw. in unserem Kurs 'de.uni-hd.deinname.productname'. Der Bundle Identifier identifiziert eine App eindeutig im App Store und auf Apple's Servern.

**Class Prefix** wird den Dateinamen aller erstellten Dateien vorangestellt (*s. S. 43, Abschnitt 3.3*).

**Devices** gibt die Möglichkeit, die App für iPhone, iPad oder Universal zu konfigurieren.

## 2 Xcode

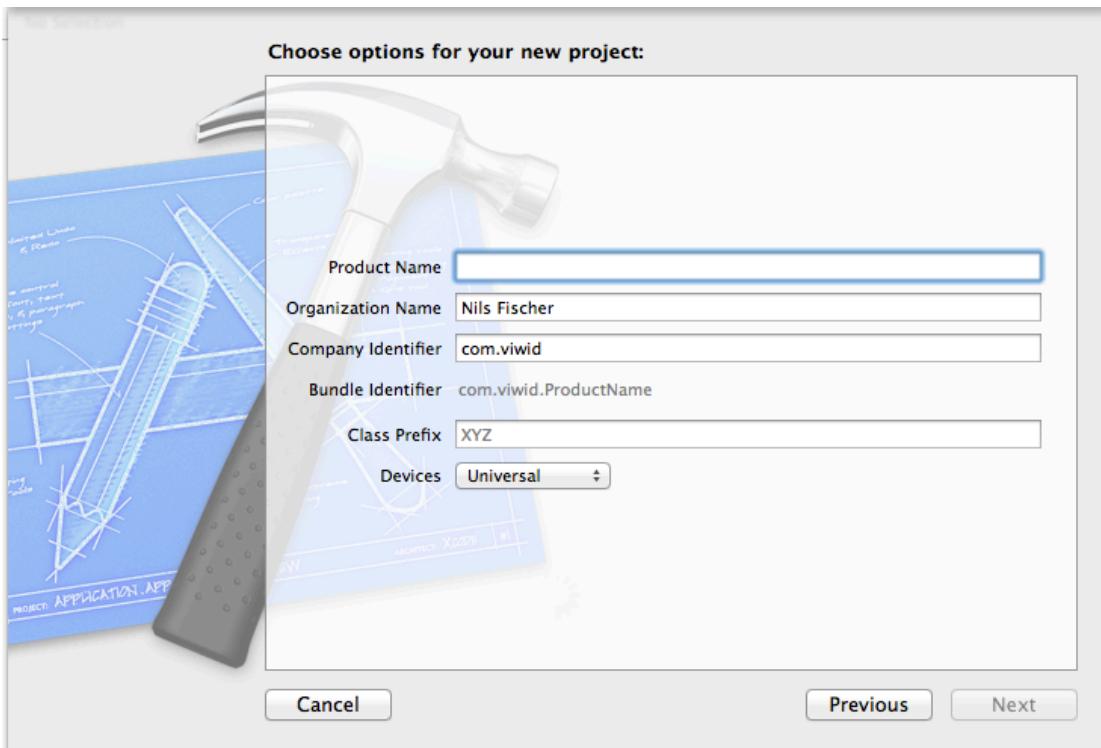


Abbildung 2.2: Ein neues Projekt konfigurieren

**Use Core Data** fügt dem Projekt direkt die benötigten Elemente zur Core Data Integration hinzu. Dies ist eine Datenbanktechnologie auf Basis von SQL, mit der wir uns noch genauer befassen werden.

Anschließend kann ein Speicherort für den Projektordner gewählt werden. Hier besteht zusätzlich die Möglichkeit, direkt ein Git Repository für das Projekt anzulegen. Git werden wir zu einem späteren Zeitpunkt noch thematisieren.

### 2.1.2 Targets und Products

Ein Projekt kann zur Entwicklung mehrerer Apps dienen, die einen direkten Bezug zueinander haben und sich meist einen Großteil des Programmcodes teilen. Dazu gehören vor allem Testversionen mit reduziertem Funktionsumfang oder einzelne Appversionen für iPhone und iPad, wenn man sich gegen die empfohlene Universal-Methode entscheidet.

Das Ergebnis des Compilers, also unsere fertige App, wird **Product** genannt. Zu jedem Product gehört genau ein **Target**. Das Target stellt die Repräsentation des Products in Xcode dar und gibt dem Compiler Auskunft über alle Konfigurationen, referenzierten Dateien und sonstige Informationen die zur Kompilierung benötigt werden.

Wenn wir ein neues Projekt erstellen, wird automatisch ein Target mit den gegebenen Informationen generiert. Mit **New > Target** kann außerdem jederzeit ein neues Target hinzugefügt und konfiguriert werden. Jedes Target muss einen eindeutigen Bundle Identifier besitzen. Anschließend können wir die Targets separat kompilieren und so jeweils ein Product erhalten.

### 2.1.3 Projekt- und Target-Konfiguration

Wenn das Projekt selbst im Project Navigator (*s. S. 14, Abschnitt 2.2.3*) ausgewählt ist, wird im Editor-Bereich die Projekt- und Target-Konfiguration angezeigt. Mit der Schaltfläche oben links kann das Projekt oder ein Target ausgewählt werden.

Hier können alle wichtigen Einstellungen bearbeitet werden, die die App als Ganzes betreffen. Wählen wir ein Target aus, kann aus den Tabs in der oberen Leiste gewählt werden:

#### General

Hier ist eine Auswahl wichtiger Optionen zur Konfiguration unserer App zu finden, von denen viele direkt mit dem jeweiligen Eintrag im Tab 'Info' korrespondieren.

**Bundle Identifier** wurde bereits besprochen und kann hier verändert werden. Wenn der Bundle Identifier einen hellgrauen, nicht editierbaren Teil enthält (meist der Product Name), dann wird dieser Teil aus einer Variable entnommen. Er kann dann im Tab 'Info' editiert werden. Verwendet bitte das Schema 'de.uni-hd.deinname.productname' für Apps, die unserem Developer Team zugeordnet sind (*s.u.*).

**Version/Build** gibt die aktuelle Versionsnummer an.

**Team** bestimmt die Zugehörigkeit der App zu einem Developer Team, sodass Xcode das richtige Provisioning Profile zur Signierung der App auswählen oder ein passendes Provisioning Profile erstellen kann (*s. S. 24, Abschnitt 2.5*).

**Deployment Target** bestimmt die iOS Version, die für die Installation der App auf dem Zielsystem **mindestens** installiert sein muss. Zusätzlich gibt es im 'Info' Tab die **Base SDK** Einstellung. Diese gibt an, für welche iOS Version die App kompiliert wird. Letztendlich bedeutet dies: Im Code können nur Features verwendet werden, die in der Base SDK Version (meist die neueste iOS Version) enthalten sind. Ist die Deployment Target Version geringer (um auch ältere Geräte zu unterstützen), so muss aufgepasst werden, dass bei neueren Features im Code immer zuerst deren Verfügbarkeit geprüft wird.

**Devices** gibt an, ob die App nur für iPhone **oder** iPad oder Universal für beide Geräte entwickelt wird.

**Main Interface** bestimmt die Interface-Datei, die beim Starten der App geladen wird. Diese Option hat weitreichende Auswirkungen auf die initiale Startsequenz der App, die wir noch thematisieren werden. In den meisten Fällen sollte hier die Storyboard-Datei für iPhone bzw. iPad ausgewählt werden.

**App Icons / Launch Images** referenziert die jeweiligen Bilddateien. Diese sollten für optimale Performance in einem sog. **Asset Catalog** zusammengefasst werden.

### Capabilities

Einige häufig verwendete Features von iOS Apps, für deren Verwendung ansonsten einige Konfigurationsschritte notwendig wären, können hier einfach aktiviert werden. Dazu gehören bspw. Services wie GameCenter, iCloud und In-App-Purchase. Wird die Schaltfläche rechts aktiviert, werden die benötigten Konfigurationen im Projekt vorgenommen. Alle Veränderungen, die bei der Aktivierung des Features ausgeführt werden, sind hier aufgeführt.

### Info

Dieser Tab ist hauptsächlich eine Repräsentation der 'Info.plist'-Datei. Alle Einstellungen, die nicht den Compiler betreffen, sondern dem ausführenden Gerät zur Verfügung gestellt werden, werden in dieser Datei gespeichert. Die meisten Optionen im 'General'-Tab verändern direkt die Einträge dieser Datei. Mittlerweile muss hier nur noch selten manuell etwas geändert werden.

### Build Settings

Hier wird der Compiler konfiguriert. Beispielsweise kann der Product Name hier verändert werden und die Base SDK Version angepasst werden, wenn nicht für 'Latest iOS', sondern eine vorherige iOS Version kompiliert werden soll. Auch die hier zu findenden Optionen sollten mittlerweile nur noch selten benötigt werden.

## 2.2 Benutzerinterface

Xcode's Interface ist in die in der Abbildung farbig markierten Bereiche aufgeteilt (s. S. 13, Abb. 2.3).

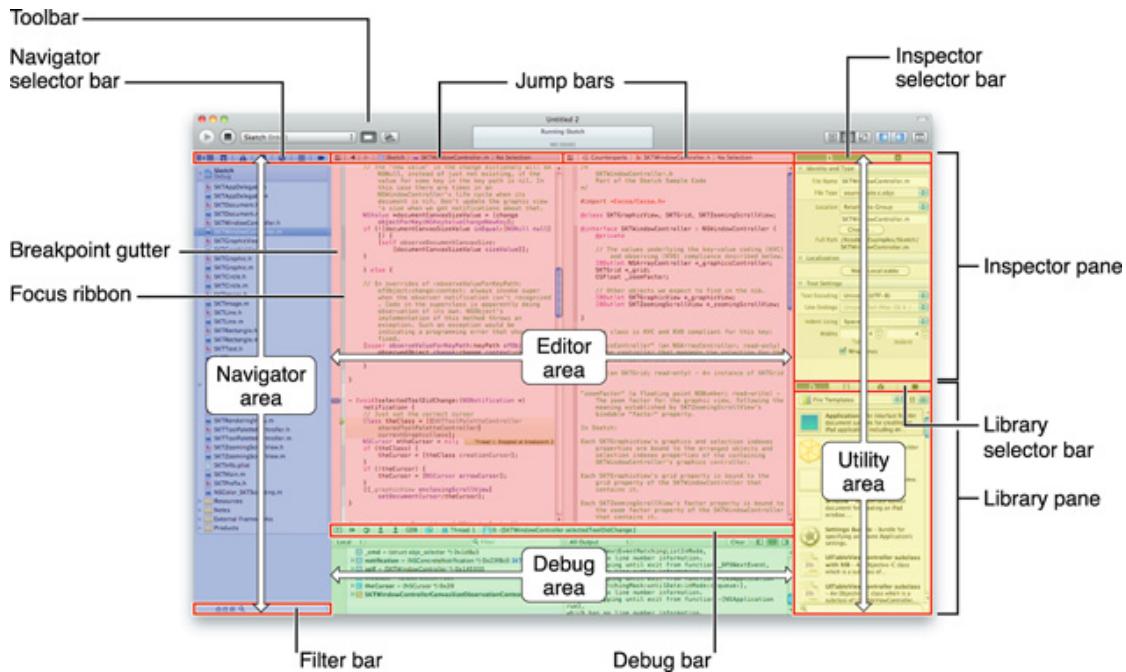


Abbildung 2.3: Xcode's Interface

### 2.2.1 Darstellungsmodi

Rechts in der Toolbar können wir mit sechs Bedienelementen die Darstellung des Xcode-Fensters anpassen.

Die ersten drei Buttons beziehen sich auf den Editor:

**Standard-Editor** zeigt einen großen Editor-Bereich zur Betrachtung einer einzelnen Ansicht an.

**Assistant-Editor** teilt den Editor-Bereich in zwei Ansichten. Auf der linken Seite befinden sich die geöffnete Datei, während die rechte Seite eine sinnvolle zugehörige Ansicht zeigt. Wir verwenden hauptsächlich diese Option und werden noch lernen, sie zu verwenden.

**Version-Editor** ersetzt den Assistenten auf der rechten Seite mit einer Ansicht der Änderungshistorie der Datei links. Verwendet das Projekt ein Git Repository, werden hier die Commits angezeigt, die die Datei betreffen. Bezuglich Git und Versionskontrolle wird es später eine Einführung geben.

Mit den weiteren drei Buttons können die Bereiche 'Navigator', 'Debug' und 'Inspector' ein- und ausgeblendet werden.

Es können ebenfalls mit  $\text{⌘} + \text{T}$  weitere (browserähnliche) Tabs geöffnet werden.

### 2.2.2 Build & Run

In der Toolbar finden wir links die Bedienelemente 'Build & Run', 'Stop' und eine Targetauswahl.

Hier kann das zu kompilierende Target und das Zielsystem ausgewählt werden. Haben wir ein gültiges iOS Gerät angeschlossen, wird dieses an erster Stelle angezeigt, andernfalls erscheint 'iOS Device' und wir können einen iOS Simulator auswählen.

Mit 'Build & Run' starten wir den Compiler, woraufhin das gewählte Target kompiliert und das resultierende Product, also die App, auf dem gewählten Zielsystem ausgeführt wird. 'Stop' beendet den Prozess. Im Menü **Product** stehen noch weitere Optionen zur Verfügung. Da wir diese sehr häufig verwenden werden ist es sinnvoll, sich die Tastenkombinationen einzuprägen:

**Build** **⌘+B** Startet den Compiler, ohne dass das Product anschließend ausgeführt wird. Diese Option ist hilfreich, um kurz die Ausführbarkeit des Targets zu prüfen und Fehler zu korrigieren.

**Build & Run** **⌘+R** Kompiliert das Target und führt das resultierende Product auf dem Zielsystem aus.

**Stop** **⌘+.** Beendet den aktiven Prozess.

**Clean** **⌘+⇧+K** Entfernt kompilierte Build-Dateien und führt zu einer vollständigen Neukompilierung beim nächsten 'Build' Aufruf. Da Xcode bereits verarbeitete Dateien wiederverwendet, solange sie nicht verändert wurden, löst diese Option manchmal Probleme, wenn Dateien außerhalb von Xcode bearbeitet wurde (bspw. Bilddateien).

**Archive** Kompiliert das Target und erstellt ein Archiv. Dieses kann anschließend verwendet werden, um die App an Tester zu verteilen oder im App Store zu veröffentlichen.

### 2.2.3 Navigator

Der **Navigator** (blau) dient zur Übersicht über die Projektelemente. Es kann zwischen acht Tabs (sieben vor Xcode Version 5.0) gewählt werden, die über die Tastenkombinationen **⌘+1** bis **⌘+8** erreichbar sind:

1. '**Project Navigator
- 2. '**Symbol Navigator
- 3. '**Find Navigator
- 4. '**Issue Navigator
- 5. '**Test Navigator**********

6. 'Debug Navigator': Übersicht der Situation, wenn eine App läuft oder zur Laufzeit angehalten wird
7. 'Breakpoint Navigator': Liste der Breakpoints
8. 'Log Navigator': Liste der letzten Output- und Compiler-Logs

Wir verwenden hauptsächlich den Project Navigator, um zwischen den Dateien unseres Projekts zu wechseln. Der Find Navigator bietet sowohl eine projektübergreifenden Suche als auch eine sehr hilfreiche 'Find & Replace' Funktion. Zur Laufzeit einer App wird der Debug Navigator wichtig, der sowohl einige Geräteinformationen anzeigt (bspw. CPU- und Speicherauslastung), als auch eine Übersicht über die laufenden Operationen, wenn die App angehalten wird.

## 2.2.4 Editor

Der **Editor** (rot) wird je nach geöffnetem Dateityp den Editor zum Bearbeiten der jeweiligen Datei zeigen.

Hier schreiben wir unseren Code. Es stehen viele hilfreiche Funktionen zur Verfügung, mit denen das Schreiben effizienter wird und Fehler schon vor dem Kompilieren erkannt und korrigiert werden können.

### Autovervollständigung

Xcode indexiert sowohl Apple's Frameworks als euren eigenen Code und besitzt somit ein umfassendes Verständnis der verwendeten Symbole. Sobald du zu tippen beginnst, werden Vorschläge eingeblendet, die dem aktuellen Kontext entsprechen (*s. S. 16, Abb. 2.4*). Die Indexierung ist so vollständig, dass nahezu kein Objective-C Codesymbol komplett ausgeschrieben wird. Stattdessen kannst du meist nach den ersten Buchstaben beginnen, die Vervollständigung zu nutzen. Dabei werden folgende Tasten verwendet:

**Escape** Blendet die Vorschläge aus oder ein.

**Tab** Vervollständigt das Symbol bis zur nächsten uneindeutigen Stelle

**Enter** Vervollständigt das gesamte Symbol.

In den meisten Fällen gibt es ein Symbol nicht, wenn es von der Autovervollständigung nicht vorgeschlagen wird! Das betrifft auch selbstgeschriebenen Code.

Objective-C ist eine sehr deskriptive Programmiersprache, deren Symbole nach Konventionen benannt sind (*s. S. 43, Abschnitt 3.3*). So können mit etwas Übung und Hilfe der Autovervollständigung auch unbekannte Symbole aus Apple's Frameworks gefunden werden, ohne erst die Dokumentation zu durchsuchen. Suchen wir beispielsweise eine bestimmte Konstante, die das Verhalten einer Animation eines **UIView**-Objekts bestimmt, so ist es typisch, die Autovervollständigung folgendermaßen zu verwenden:

## 2 Xcode

```
63
64 - (void)application:(UIApplication *)application didChangeStatusBarFrame:(CGRect)oldStatusBarFrame
65 - (void)application:(UIApplication *)application didChangeStatusBarFrame:(CGRect)oldStatusBarFrame
66 - (void)application:(UIApplication *)application didChangeStatusBarOrientation:(UIInterfaceOrientation)oldStatusBar...
67 - (void)application:(UIApplication *)application didDecodeRestorableStateWithCoder:(NSCoder *)coder
68 - (void)application:(UIApplication *)application didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
69 - (void)application:(UIApplication *)application didReceiveLocalNotification:(UILocalNotification *)notification
70 - (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo
71 - (void)application:(UIApplication *)application didReceiveRemoteNotification:(NSDictionary *)userInfo fetchComple...
72 - (void)application:(UIApplication *)application didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceT...
73
74
75
76
77 } Tells the delegate when the frame of the status bar has changed. More...
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
```

Abbildung 2.4: Der meiste Code wird mit der Auto vervollständigung geschrieben

1. Wir beginnen mit dem Tippen von **UIVi**, verwenden die Tab-Taste, um zur nächsten uneindeutigen Stelle zu springen und erhalten **UIView**.
  2. In der Liste der Vorschläge sehen wir unter anderem Symbole, die mit **UIViewAnimation** beginnen. Mit den Pfeiltasten wählen wir eines aus und drücken wieder Tab.
  3. In dieser Weise ist es sehr einfach, die möglichen Optionen der Animation zu finden, ohne sie zuvor zu kennen (s. S. 16, Abb. 2.5). Mit den Pfeiltasten können wir die gewünschte Option auswählen und mit der Enter-Taste einfügen.

```
[UIView animateWithDuration:2.0 delay:0 options:UIViewAnimationOptionCurveEaseInOut animations:^{
    self.view.alpha = 0;
} completion:^(BOOL finished) {
    self.view.alpha = 1;
}];

[super viewDidLoad];
}

- (void)viewDidLoad {
    [super viewDidLoad];
    // for shake gesture
    [self becomeFirstResponder];
}
```

Abbildung 2.5: Auch unbekannte Symbole können mit der Autovervollständigung gefunden werden

## Fehlerkorrektur

Viele häufig auftretende Syntaxfehler werden schon bei der Codeeingabe von Xcode erkannt und können sofort korrigiert werden (*s. S. 17, Abb. 2.6*). Dazu gehören fehlende Steuerzeichen wie Semikolons, aber auch komplexere Fehler.

## 2 Xcode

A screenshot of the Xcode code editor. A specific line of code is highlighted with a red oval, indicating a syntax error. The line contains a call to the `shuffle` method. A callout bubble appears over the code, containing the message "Issue Expected ';' after expression" and a "Fix-it" button with the suggestion "Insert ";"".

```
68 }      // ...
69 if (!_pack) {
70     // create new
71     _pack = [[Pack alloc] init];
72     _pack.packCount = kDefaultPackCount;
73     [_pack shuffle]; // Issue Expected ';' after expression
74 }
75 return _pack; // Fix-it Insert ";"
```

Abbildung 2.6: Xcode's Fehlerkorrektur erkennt und behebt Syntaxfehler

### Integrierte Dokumentation & Links

Mit einem ⌘-Klick auf ein Symbol im Code kann jederzeit eine kurze Definition desselben angezeigt werden, sofern es in der Dokumentation enthalten ist (s. S. 17, Abb. 2.7).

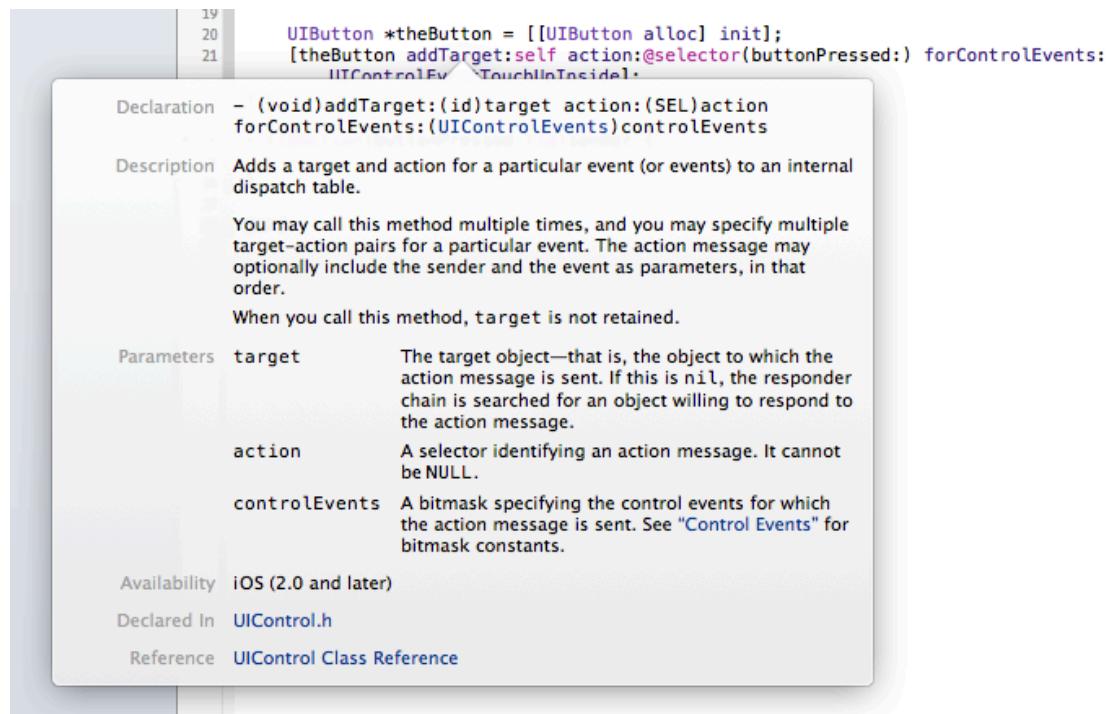


Abbildung 2.7: Alt-Klick auf ein Symbol zeigt eine kurze Definition

Ein ⌘-Klick wirkt wie ein Link im Internet und führt je nach Kontext direkt zur Deklaration des Symbols im Projekt oder zum Ziel des Aufrufs.

## Jump bars & Open Quickly

Die Leiste oben im Editor-Bereich (genannt Jump bar) dient der Navigation und zeigt den Pfad der geöffneten Datei im Projekt an. Mit einem Klick auf ein Pfadsegment kann auf die Dateistruktur zugegriffen werden. Sehr praktisch ist, dass hier sofort etwas getippt werden kann, woraufhin die Liste gefiltert wird. Das gilt auch für das letzte Pfadsegment, das die Position im Code der geöffneten Datei anzeigt und der schnellen Navigation innerhalb deren Symbole dient.

Tipp: Ist im Code an einer Stelle der Ausdruck `#pragma mark Section Title` zu finden, so erscheint 'Section Title' als Gliederung in dem Menü der Jump bar. Ein Gedankenstrich erzeugt eine horizontale Linie: `#pragma mark – Section Title`.

Mit der 'Open Quickly' Funktion  +  +  kann ebenfalls schnell navigiert werden. Es öffnet sich ein Eingabefeld, mit dem auf die gesamte Indexierung des Projekts zugegriffen werden kann.

## Assistent

Im Assistant-Mode wird der Editor-Bereich zweigeteilt. Während der linke Teil die geöffnete Datei anzeigt, kann im rechten Teil mit Klick auf das erste Segment der Jump bar eine zugehörige Datei geöffnet werden. Empfehlenswert ist hier die Option 'Counterpart', die zu einer Main-Datei immer die entsprechende Header-Datei anzeigt (*s. S. 37, Abschnitt 3.2.2*). Diese Ansicht werden wir hauptsächlich verwenden, es stehen jedoch noch weitere situationsbedingte Optionen zur Verfügung.

## Breakpoints

Mit einem Klick auf eine Zeilennummer in der Leiste rechts vom Editorbereich kann ein Breakpoint in dieser Codezeile gesetzt werden, sodass die App bei der Ausführung an dieser Stelle angehalten wird (*s. S. 19, Abschnitt 2.2.6*).

## 2.2.5 Inspektor

Am rechten Bildschirmrand kann der Inspektor eingeblendet werden, dessen Tabs ähnlich wie beim Navigator mit den Tastenkombinationen  +  +  bis  +  +  erreichbar sind. Während Code geschrieben wird, sind hier nur zwei Tabs verfügbar:

1. 'File Inspector': Optionen bezüglich der im Editor geöffneten Datei
2. 'Quick Help Inspector': Kurze Dokumentation des ausgewählten Symbols im Editor, ähnlich den Informationen, die durch  - Klick auf das Symbol erreichbar sind

Zur Codeeingabe wird der Inspektor meist ausgeblendet, doch für die Konfiguration von Benutzeroberflächen mit dem Interface Builder ist er unverzichtbar (s. S. 20, Abschnitt 2.3).

### 2.2.6 Debug-Bereich & Konsole

Der Debug-Bereich im unteren Bildschirmbereich wird zur Laufzeit einer App verwendet.

In der Konsole werden Ausgaben angezeigt, die von der App generiert werden. Wir werden noch lernen, diese zu nutzen, um den ausgeführten Code während der Laufzeit nachzuvollziehen. Außerdem wird hier die exakte Situation der App angezeigt, wenn diese zur Laufzeit angehalten wird (s. S. 19, Abb. 2.8).

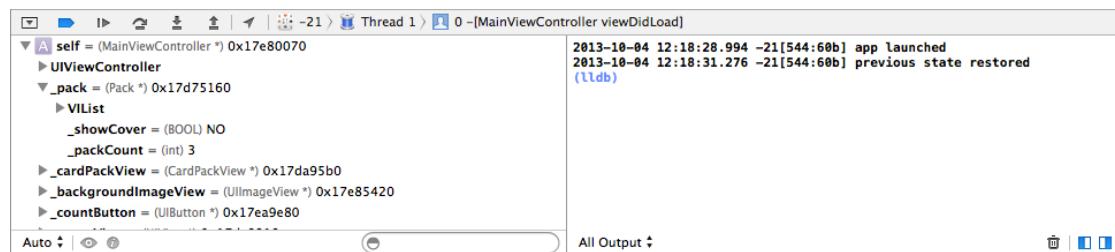


Abbildung 2.8: Im Debugger werden Konsolenausgaben und Situation der App angezeigt

Die beiden Bereiche können mit den beiden Schaltern in der rechten unteren Ecke umgeschaltet werden.

In der Leiste im oberen Teil des Debug-Bereichs sind folgende Kontrollelemente zu finden:

**Breakpoints** aktiviert/deaktiviert die Breakpoints im gesamten Projekt (s. S. 18, Abschnitt 2.2.4).

**Pause/Resume** stoppt die Ausführung App oder startet diese wieder.

**Step over** führt die im Editor markierte Codezeile aus.

**Step into** folgt dem im Editor markierten Ausdruck, bspw. einem Methodenaufrufen.

**Step out** führt den aktuellen Methodenaufruf vollständig aus und zeigt ihn anschließend an.

Während die App angehalten ist, können im Debug-Bereich und im Editor-Bereich relevante Symbole inspiziert werden. Fährt man mit dem Cursor über ein Symbol, können Informationen über den aktuellen Status desselben angezeigt und in der Konsole ausgegeben werden. Zusätzlich zu herkömmlichen Werten der Variablen können sogar Bilder, die in den ausführenden Speicher geladen wurden, mit Quicklook angezeigt werden.

Da Code selten sofort so funktioniert wie wir möchten, ist Debugging eine wichtige Komponente der Programmierung. Wir werden uns daher noch genauer mit den verschiedenen Methoden beschäftigen um Fehler im Programmcode zu finden und auch die Speicherauslastung und Performance unserer Apps zu optimieren.

## 2.3 Interface Builder

So gut euer Code auch geschrieben sein mag – Benutzer werden nur die Benutzeroberfläche oder User Interface (UI) eurer Apps zu sehen bekommen. Diese ist ein wichtiger Bestandteil der iOS Plattform und wir werden uns noch mit einigen Methoden beschäftigen, sinnvoll gestaltete und dynamische UIs zu erstellen.

In Xcode ist, wie in vielen IDEs, ein graphischer Editor genannt Interface Builder (IB) integriert, der bei der UI-Gestaltung hilft. Alles, was mit dem Interface Builder erstellt wird, kann natürlich auch stattdessen in Code geschrieben werden. Doch bereits bei simplen UIs vereinfacht IB die Gestaltung um ein Vielfaches und hilft mit Konzepten wie Storyboarding und Auto Layout zusätzlich bei der Strukturierung der gesamten Benutzerführung und der dynamischen Anpassung des UI's an verschiedene Displaygrößen und -orientierungen. Für komplexere UIs werden wir die Vorteile des Interface Builders daher schnell zu schätzen lernen.

Bei Dateien mit der Endung .xib oder .storyboard wird Xcode's Editor-Bereich automatisch mit dem Interface Builder ersetzt. Wir verwenden dann meist einen neuen Tab mit angepasster Konfiguration, blenden den Navigator- und Debug-Bereich aus und den Inspektoren-Bereich ein (*s. S. 21, Abb. 2.9*). Im Editor-Bereich wird situationsbedingt der Standard- oder Assistant-Editor verwendet. Im IB-Modus wird im Editor-Bereich dann auf den linken Seite eine Navigationsleiste eingeblendet, die die Elemente in der geöffneten Datei anzeigt. Diese kann mit der Schaltfläche unten ein- und ausgeblendet werden.

Im unteren Bereich des Inspektors ist die **Object Library** zu finden. Wir können Objekte aus dieser Liste auf ein Element im Interface Builder ziehen und es so hinzufügen. Zu diesen Objekten gehören sowohl Interfaceelemente wie Buttons und Labels als auch strukturgebende Elemente wie Navigation Controller. Diese Objekte lernen wir bei der Erstellung unserer Apps noch kennen.

Anschließend können die Objekte in der Navigationsleiste links oder im Editor ausgewählt werden und mit dem Inspektor konfiguriert werden. Sind mehrere Elemente im Editor übereinander positioniert, hilft ein ⌘-Klick auf die entsprechende Stelle. Es wird eine Liste der unter dem Cursor angeordneten Objekte angezeigt, aus dem das Gesuchte ausgewählt werden kann.

## 2 Xcode

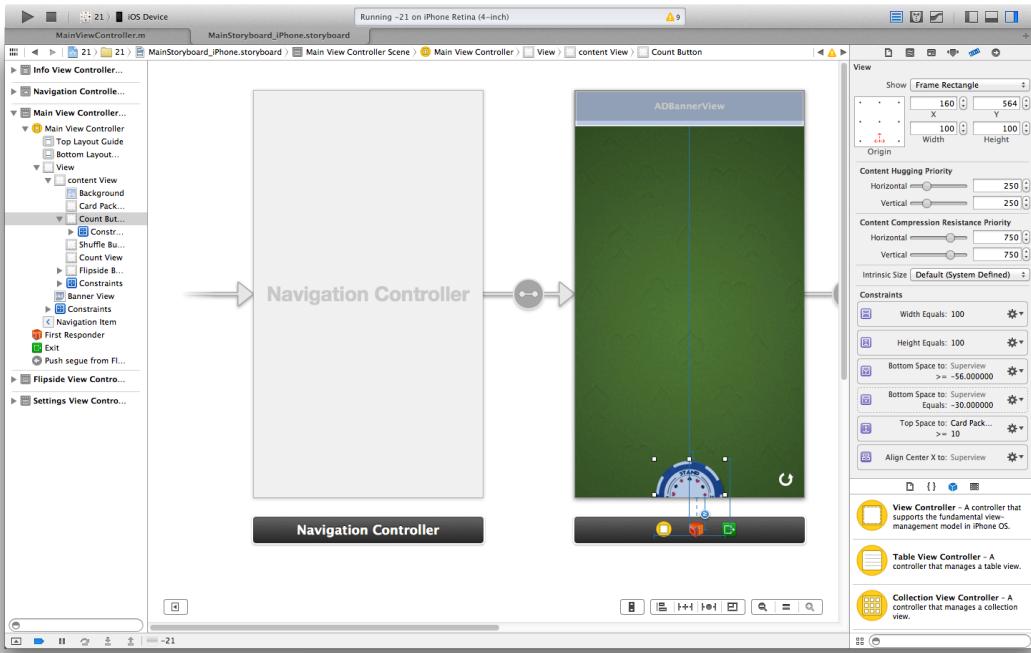


Abbildung 2.9: Um möglichst viel Platz zur UI-Gestaltung zu erhalten, verwenden wir für den Interface Builder eine angepasste Konfiguration

### 2.3.1 XIBs & Storyboards

Bevor das **Storyboarding**-Konzept eingeführt wurde, wurden für die Interfacegestaltung einzelne **.xib**-Dateien verwendet (auch aufgrund ihrer ursprünglichen Endung **NIB-Dateien** genannt). Storyboards hingegen vereinen meist die Interfaceelemente der gesamten App und auch ihre benutzerführenden Verbindungen in einer **.storyboard**-Datei.

Das UI einer App wird dann hauptsächlich in seinem Storyboard konfiguriert, während es im Code mit Inhalten gefüllt wird.

Für Universal Apps erstellen wir ein Storyboard für die iPhone Version der App und eines für die iPad Version. In den Target-Einstellungen (s. S. 11, Abschnitt 2.1.3) wird das jeweils verwendete Storyboard ausgewählt.

In einem Storyboard stellen dann einzelne **Scenes** die verschiedenen Ansichten dar, die dem Benutzer präsentiert werden. Eine der Scenes kann als **Initial Scene** gekennzeichnet werden und wird dem Benutzer zuerst präsentiert.

Zwischen den Scenes vermitteln **Segues**. Diese können eine Verbindung zwischen Scenes darstellen, bspw. wenn durch eine Benutzereingabe eine andere Scene angezeigt werden soll.

### 2.3.2 Inspektor im IB-Modus

Im IB-Modus stehen im Inspektor zusätzlich zum File- und Quick-Help-Inspektor (*s. S. 18, Abschnitt 2.2.5*) vier weitere Tabs zur Verfügung, mit denen ein ausgewähltes Objekt im Interface Builder konfiguriert wird:

**Identity Inspector** dient dem Einstellen der Identität des Objekts, also hauptsächlich seiner Klasse (*s. S. 36, Abschnitt 3.2.1*).

**Attributes Inspector** zeigt alle Konfigurationsoptionen bezüglich der Eigenschaften des Objekts nach Subklasse sortiert an. Dazu gehören bspw. Hintergrund- und Textfarben.

**Size Inspector** enthält Optionen zu Größe und Position des Objekts.

**Connections Inspector** zeigt die verbundenen IBOutlets und IBActions des Objekts an (*s. S. 22, Abschnitt 2.3.3*).

### 2.3.3 IBOutlets & IBActions

Hinweis: Für diesen Abschnitt ist Kenntnis über **Properties** (*s. S. 37, Abschnitt 3.2.3*) und **Methoden** (*s. S. 38, Abschnitt 3.2.4*) notwendig .

#### IBOutlets

Haben wir unser UI im Interface Builder konfiguriert, möchten wir häufig im Code auf die verwendeten Objekte zugreifen. Dazu verwenden wir sog. **IBOutlets**. Dies sind speziell gekennzeichnete **Properties** einer Klasse:

```
1 @property (nonatomic, strong) IBOutlet UILabel *label; // diese Property ist
   als IBOutlet gekennzeichnet
```

In der XIB oder dem Storyboard können wir dann eine Verbindung zwischen dem Objekt und dem IBOulet herstellen (*s. S. 22, Abb. 2.10*).

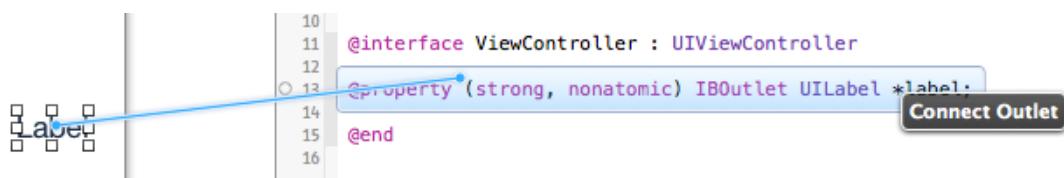


Abbildung 2.10: IBOutlets verbinden Interfaceelemente mit Properties im Code

Zur Laufzeit der App wird dieser Property dann das verbundene Objekt als Wert zugewiesen, sodass im Code darauf zugegriffen werden kann. Hätten wir das Objekt im Code erstellt, wäre eine IBOulet-Verbindung also äquivalent zu folgendem Code:

```
1 self.label = theLabel; // Sei theLabel das zuvor erstellte Objekt
```

Um sehr einfach IBOutlets zu erstellen, wechseln wir in den Assistant-Editor. Im Editor Bereich wird dann rechts der Assistent angezeigt. Hier können wir oben in der Jump bar **Automatic > Klassename.h** wählen, also die zugehörige Header-Datei, in der die Property deklariert wurde. Mit gedrückter **ctrl**-Taste können wir nun eine Verbindung zwischen dem Interfaceelement und der Property ziehen (s. S. 22, Abb. 2.10). Häufig sollen solche Properties nicht öffentlich sein, da nur innerhalb der Klasse auf die Interfaceelemente zugegriffen wird. Dann sollten sie im privaten Interface in der Main-Datei definiert werden (s. S. 37, Abschnitt 3.2.2).

Alternativ kann der Connection Inspector des Objekts (s. S. 22, Abschnitt 2.3.2) oder ein Rechtsklick auf das Objekt verwendet werden und ausgehend von dem Kreissymbol neben 'New Referencing Outlet' eine Verbindung gezogen wird. Wird die Verbindung nicht zu einem existierenden IBOutlet im Code, sondern auf eine leere Zeile gezogen, wird automatisch eine mit 'IBOutlet' gekennzeichnete Property erstellt. Anstatt den Assistant-Editor zu verwenden kann das Ziel der Verbindung auch im Interface Builder gesucht werden, also bspw. in der Navigationsleiste links.

## IBActions

**IBActions** funktionieren ähnlich wie IBOutlets und stellen eine Verbindung zu **Methoden** einer Klasse her.

Einige Objekte stellen sog. **Events** zur Verfügung, die in bestimmten Situationen ausgelöst werden. Dazu gehören bspw. Subklassen von UIControl, also u.a. UIButton, die das Event 'Touch Up Inside' auslösen, wenn der Benutzer seinen Finger im Bereich des Objekts anhebt.

Diese Events können mit Methoden im Code verbunden werden, die mit 'IBAction' gekennzeichnet sind (s. S. 23, Abb. 2.11).

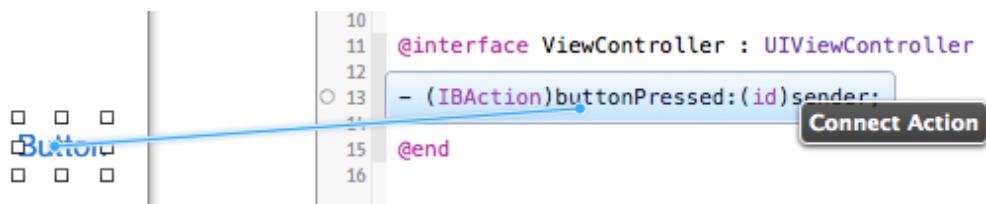


Abbildung 2.11: IBActions verbinden Events mit Methoden im Code

Die verbundene Methode wird dann ausgeführt, wenn das Event ausgelöst wird. Das auslösende Objekt wird der Methode als Parameter `sender` des Typs `id` (beliebiger Typ) übergeben.

Die Zuweisung einer IBAction-Verbindung ist dann äquivalent zu folgendem Code:

```

1 [theButton addTarget:theReceiver action:@selector(buttonPressed:)
    forControlEvents:UIControlEventTouchUpInside]; // Sei theButton das das
    Event auslösende Objekt und theReceiver das Objekt, das die Methode -(void)buttonPressed:(id)sender implementiert

```

## 2.4 Dokumentation

Die Dokumentation von Apple's Frameworks ist exzellent mit Xcode verknüpft und sollte bei Unklarheiten immer als erste Referenz verwendet werden. In der immer verfügbaren Kurzdefinition per ⌘-Klick auf ein beliebiges Symbol im Code, ist immer ein Verweis auf die ausführliche Dokumentation enthalten (s. S. 17, Abschnitt 2.2.4). Ein Klick darauf öffnet den entsprechenden Abschnitt der Dokumentation im separaten Documentation-Fenster (s. S. 25, Abb. 2.12).

Dieses ist außerdem immer mit dem Tastenkürzel ⌘+⌘+? (bzw. ⌘+⌘+⇧+B auf deutschen Tastaturen) erreichbar.

In dem Such-Eingabefeld oben kann selbstverständlich nach beliebigen Symbolen gesucht werden, während mit den Schaltflächen rechts davon die beiden Seitenleisten 'Bookmarks' und 'Overview' ein- und ausgeblendet werden können.

Die Dokumentation ist außerdem online in der iOS Developer Library<sup>[1]</sup> verfügbar. Eine Google-Suche nach dem Klassen- oder Symbolnamen führt meist direkt dorthin. Apple bietet online zusätzlich noch viele weitere Ressourcen für iOS Developer an (s. S. 7, Abschnitt 1.5).

## 2.5 Testen auf iOS Geräten

Der Simulator eignet sich sehr gut zum Testen eurer Apps. Er läuft auf der Architektur eures Mac's und ist daher in den meisten Fällen deutlich schneller als ein iOS Gerät, verfügt jedoch nicht über alle Funktionen eines solchen. Außerdem gibt es einen großen Unterschied zwischen der Bedienung einer App mit Touchpad oder Maus im Vergleich zu einem Touchscreen. Beispielsweise wird die Größe und Position von Schaltflächen beim Testen auf dem Simulator häufig falsch eingeschätzt, da ein Finger auf dem Touchscreen sehr viel ungenauer tippen kann als mit dem Cursor geklickt wird und häufig Teile des Bildschirms verdeckt (s. S. 26, Abb. 2.13).

Auf dem Simulator kann somit kein wirklichkeitsgetreues Testen stattfinden und es ist unmöglich, Apps auch auf einem realen iOS Gerät zu testen.

---

<sup>[1]</sup><https://developer.apple.com/library/ios/>

## 2 Xcode

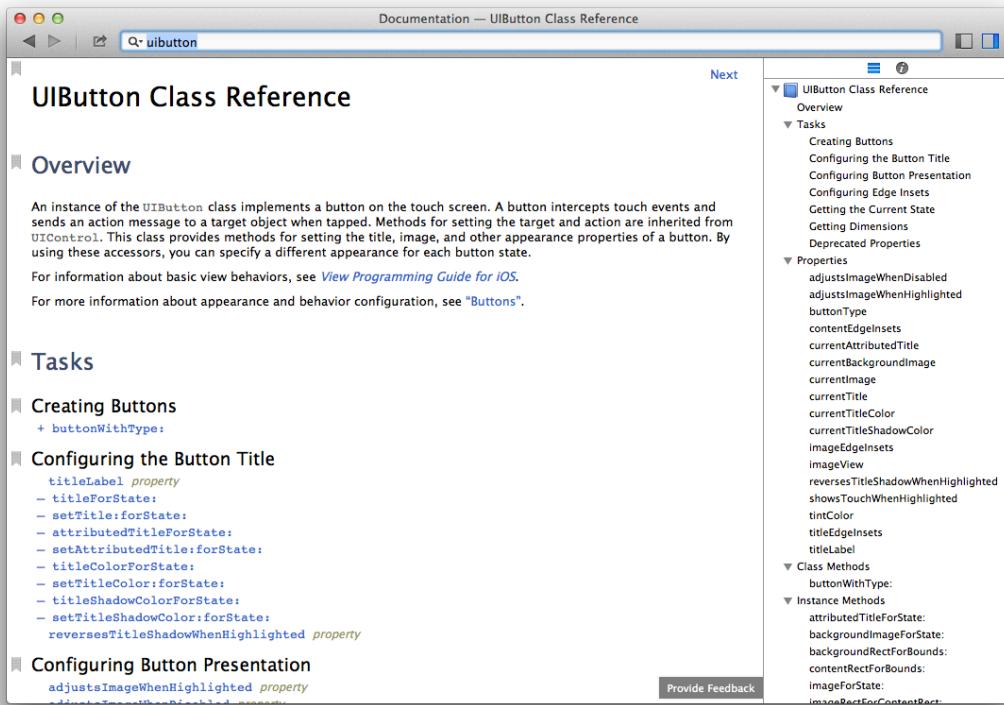


Abbildung 2.12: Xcode's integrierte Dokumentation ist beispielhaft strukturiert und sollte immer als erste Referenz verwendet werden

Es wird empfohlen, nur iOS Geräte zum Testen zu verwenden, die allein zu diesem Zweck zur Verfügung stehen, da selbstverständlich Programmierfehler auftreten können. Dafür reicht natürlich nicht immer das Budget.

Xcode kann Apps nur auf Geräten installieren, deren Softwareversion mit der jeweiligen Version von Xcode kompatibel ist. Bei neueren Versionen von Xcode lassen sich unter **Preferences** → **Downloads** ältere iOS SDKs zusätzlich installieren, um diese Versionen zu unterstützen und als Base SDK (*s. S. 11, Abschnitt 2.1.3*) zu verwenden. Die Umkehrung gilt jedoch nicht: Geräte mit neueren Softwareversionen als die installierte iOS SDK Version können nicht zum Testen verwendet werden.

Für diesen Kurs ist es empfehlenswert, die neueste Version von Xcode, dem iOS SDK und der auf dem Gerät installierten Software zu verwenden (*s. S. 6, Abschnitt 1.3*). Leider ist auf den Macs im Medienzentrum nur die Xcode Version 4.2 mit iOS SDK 5.0 installiert. Daher können mit diesen Macs keine Geräte mit einer neueren Softwareversion als iOS 5.0 zum Testen verwendet werden.



Abbildung 2.13: Unter einer Schaltfläche positionierte Interfaceelemente sind bei der Bedienung meistens vom Finger bedeckt. Solche Probleme werden häufig erst beim Testen auf realen iOS Geräten entdeckt.

### 2.5.1 Der Provisioning Prozess

Hinweis: Dieser Abschnitt stellt die Grundlagen des Provisioning Prozesses dar. Xcode vereinfacht diesen Vorgang mittlerweile sehr (s. S. 27, *Abschnitt 2.5.2*).

Damit Apps auf iOS Geräten installiert werden können, müssen diese digital signiert werden. So wird sichergestellt, dass die App von einer vertrauenswürdigen, bei Apple registrierten Quelle stammen und ausführbarer Code nicht verändert wurde.

Der Developer Mac erstellt zunächst eine **Signing Identity**, die aus einem public-private-Key-Paar besteht und im Schlüsselbund (Keychain) eures Macs gespeichert wird. Mit dem public Key wird dann ein **Certificate** angefordert, das an Apple und das Developer Team übermittelt und bestätigt wird. Dieses wird anschließend ebenfalls im Schlüsselbund gespeichert. Es wird zwischen **Development Certificates**, die einen einzelnen Entwickler identifizieren und das Testen von Apps auf dessen Geräten erlauben, und **Distribution Certificates**, die der Identifikation des Development Teams und zur Veröffentlichung von Apps im App Store dienen, unterschieden. Die Certificates sind online im Member Center [<sup>2</sup>] einsehbar.

Mit einem gültigen Certificate können nun sog. **Provisioning Profiles** (Bereitstellungsprofile) erstellt werden, die zusammen mit einer App auf das ausführende Gerät geladen werden und die benötigten Informationen bezüglich der Signierung der App liefern. Eine App kann ohne ein gültiges Provisioning Profile nicht installiert werden. Es wird wieder zwischen **Development Provisioning Profiles** und **Distribution Provisioning Profiles** unterschieden.

Ein Distribution Provisioning Profile ist immer direkt auf eine Bundle ID bezogen und ermöglicht die Veröffentlichung genau dieser App mit der Bundle ID.

Für die Entwicklung können ebenfalls Development Provisioning Profiles erstellt werden, die genau auf eine Bundle ID bezogen sind. Diese werden **explicit** genannt und sind erforderlich, wenn Services wie iCloud verwendet werden, die eine exakte Identifikation benötigen. Solange dies nicht erforderlich ist, kann anstatt der Bundle ID auch ein Asterisk

<sup>2</sup><https://developer.apple.com/membercenter/>

(\*) verwendet werden. Ein solches Development Provisioning Profile wird auch **wildcard** genannt und kann für beliebige Bundle IDs verwendet werden.

Das Provisioning Profile enthält außerdem Informationen über die Geräte, die für die Installation der App autorisiert sind. Sog. **Team Provisioning Profiles** erlauben die Installation einer App auf allen Geräten, die im Developer Team registriert sind. Es können hingegen auch Provisioning Profiles erstellt werden, die auf bestimmte Geräte beschränkt sind, bspw. um geschlossene Beta-Tests durchzuführen. In jedem Fall müssen die Geräte, die verwendet werden sollen, im Member Center mit ihrer UDID registriert sein.

### 2.5.2 Provisioning mit Xcode

Mittlerweile werden die beschriebenen Schritte zur Erstellung von gültigen Provisioning Profiles größtenteils von Xcode erledigt.

Ihr benötigt zunächst einen Apple Developer Account und müsst anschließend unserem registrierten Developer Team der Uni Heidelberg beitreten.

1. Erstellt eine Apple ID [<sup>3</sup>] oder verwendet, soweit vorhanden, eure existierende Apple ID
2. Schließt euer iOS Gerät mit dem USB-Kabel an.
3. Öffnet in Xcode den **Organizer** mit **⌘ + ⌘ + 2**.
4. Wählt dort den Tab **Devices** und in der linken Seitenleiste euer Gerät aus (*s. S. 28, Abb. 2.14*).
5. Mit den Schaltflächen **Use for Development** oder **Add to Member Center**) können nur Admins des Developer Teams ihre Geräte direkt im Team registrieren. Stattdessen kopiert bitte die UDID eures iOS Geräts, also die lange Zeichenfolge mit dem Label 'Identifier'.
6. Teilt mir **eure registrierte Apple ID** und **die UDIDs aller eurer iOS Geräte** per Email [<sup>4</sup>] mit, sodass ich euch in das Developer Team einladen und die Geräte hinzufügen kann. Verwendet dazu bitte die Email, mit der ihr euch zum Kurs angemeldet habt.
7. Mit dem Einladungslink, den ihr anschließend per Email von Apple erhaltet, könnt ihr dem Developer Team beitreten. Ihr müsst euch im erscheinenden Dialog nicht noch einmal registrieren, wählt hier einfach 'Sign in' und loggt euch mit der verwendeten Apple ID ein.

---

<sup>3</sup><https://developer.apple.com/register/>

<sup>4</sup>n.fischer@stud.uni-heidelberg.de

<sup>5</sup><https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide>

## 2 Xcode

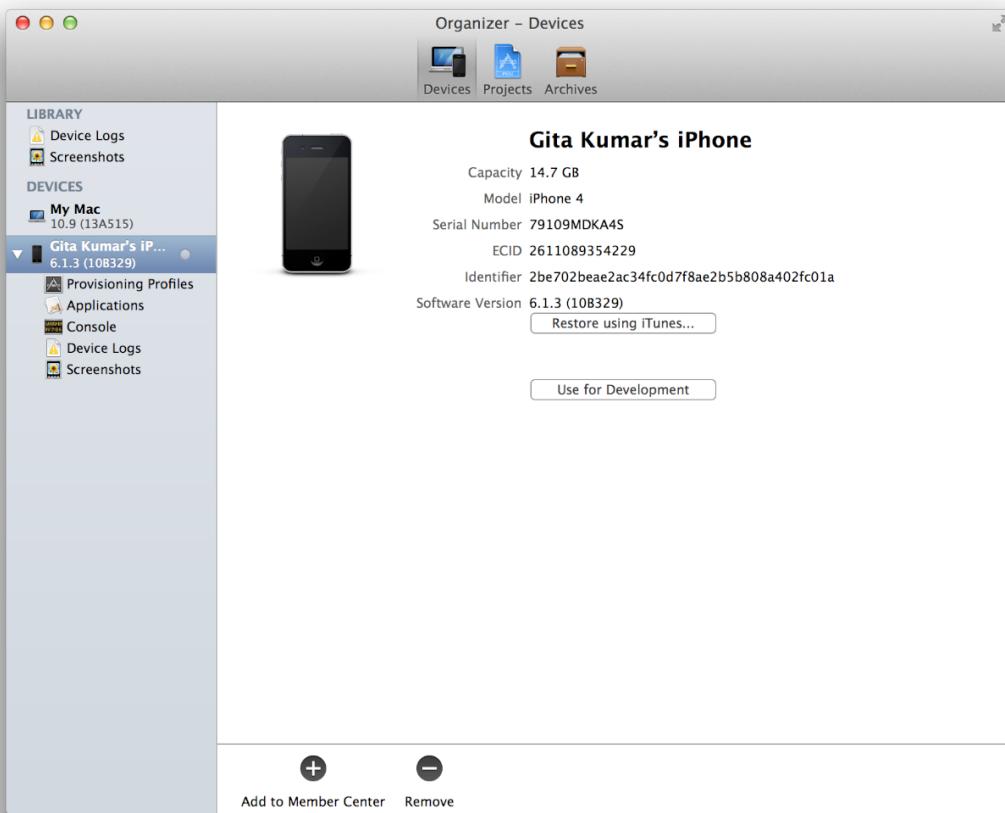


Abbildung 2.14: Im Organizer können die iOS Geräte verwaltet werden. Developer Team Admins können die Schaltfläche 'Use for Development' oder 'Add to Member Center' verwenden, andernfalls muss die UDID des Geräts manuell dem Team hinzugefügt werden. (Bild aus Apple's Dokumentation [5])

Ihr erhaltet damit Zugriff auf das iOS Dev Center [6] mit Ressourcen und Dokumentationen. Besonders hilfreich sind hier auch die Videos der jährlich stattfindenden Apple Worldwide Developer Conference (WWDC).

Nun integriert euren Apple Developer Account in Xcode:

1. Öffnet in Xcode **Preferences > Accounts** und fügt euren Apple Developer Account dort hinzu (s. S. 29, Abb. 2.15).
2. Auf der rechten Seite erscheinen die Developer Teams, denen ihr angehört. Wählt das Team der Uni Heidelberg aus und klickt **View Details**.

<sup>6</sup><https://developer.apple.com/devcenter/ios/>

3. Im erscheinenden Dialog könnt ihr nun eure Certificates und Provisioning Profiles verwalten (s. S. 30, Abb. 2.16). Die Liste der Provisioning Profiles dient hier nur zur Information. Ihr benötigt nur ein Development Certificate, das ihr mit Klick auf anfordern könnt. Es muss nun zunächst von mir bestätigt werden, daher kann zunächst eine Fehlermeldung erscheinen.
4. Ein Klick auf das -Symbol unten links aktualisiert die Liste und lädt verfügbare Certificates und Provisioning Profiles automatisch herunter. Diese sind außerdem jederzeit online im Member Center einsehbar [7].

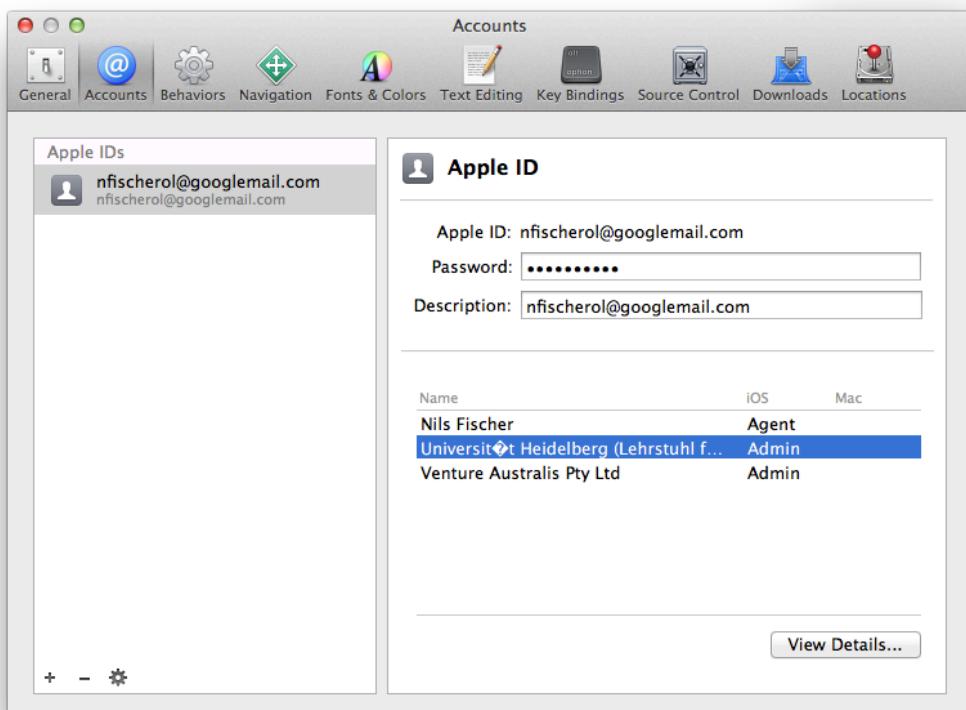


Abbildung 2.15: In den Einstellungen kann der Apple Developer Account hinzugefügt werden

Wenn Certificates oder Devices im Developer Team hinzugefügt werden, generiert Xcode automatisch auch ein wildcard Team Provisioning Profile, das euer iOS Device zum Testen von beliebigen Apps autorisiert. Außerdem werden für solche Apps, die es erfordern, zusätzlich explicit Team Provisioning Profiles generiert (s. S. 26, Abschnitt 2.5.1).

Nun könnt ihr Apps auf eurem iOS Gerät installieren und testen:

<sup>7</sup><https://developer.apple.com/membercenter/>

## 2 Xcode

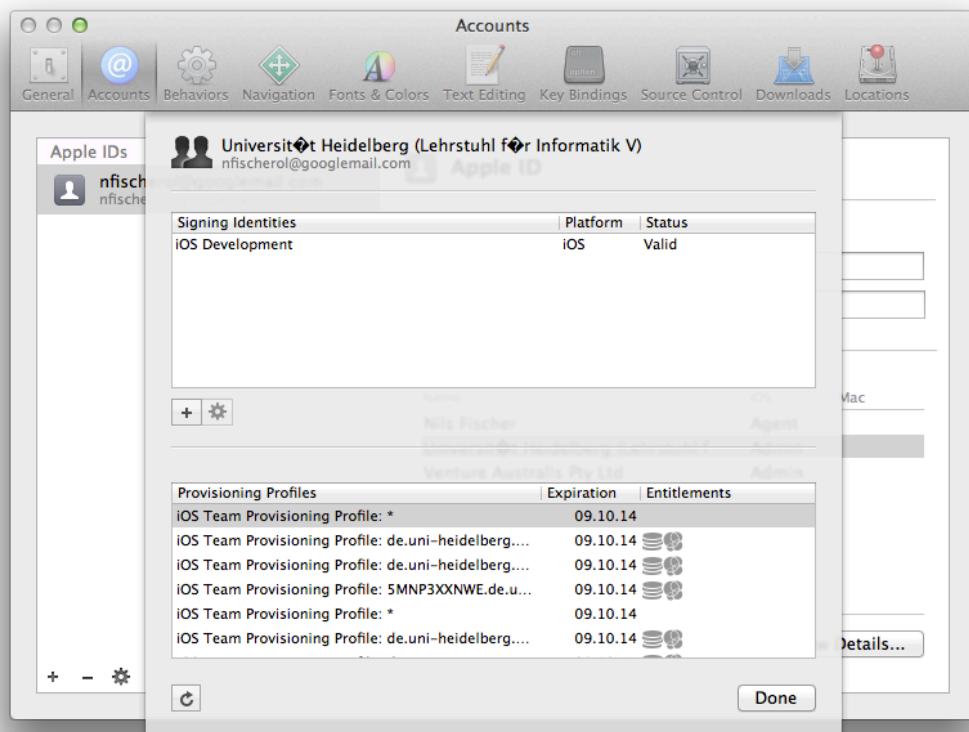


Abbildung 2.16: Mit Xcode könnt ihr eure Certificates und Provisioning Profiles verwalten

1. Stellt sicher, dass in der Target-Konfiguration unser Development Team ausgewählt ist und die Bundle ID dem beschriebenen Schema entspricht (s. S. 11, Abschnitt 2.1.3).
2. Wählt euer iOS Device als Zielsystem und führt einen 'Build & Run' aus (s. S. 14, Abschnitt 2.2.2).
3. Die App wird nun auf dem angeschlossenen Gerät ausgeführt!

# 3 Objective-C

In diesem Kurs lernen wir zunächst die Grundlagen der objektorientierten Programmierung und werden dann sehr schnell anfangen, unsere ersten iOS Apps zu schreiben. Dabei lernen wir im Verlauf des Kurses viele wichtige und allgemeingültige Konzepte und Methoden der Programmierung und Projektstrukturierung kennen, die auch problemlos auf andere Plattformen und Programmiersprachen übertragen werden können.

Software für die iOS und Mac Plattformen wird fast ausschließlich in Objective-C geschrieben. Objective-C ist eine auf C basierende, objektorientierte Programmiersprache, die in den letzten Jahren großen Zulauf erhalten hat und sehr ähnlich zu C++ ist.

Im Unterschied zu C++ wird in Objective-C vieles zur Laufzeit anstatt bei der Kompilierung ausgeführt, was auf den Fokus auf benutzergesteuerte Programmelemente zurückzuführen ist. Außerdem wird viel Wert auf eine einfache und lesbare Code-Syntax gelegt, die wir im Folgenden noch kennenlernen werden.

## 3.1 Grundlagen der Programmierung

Programmieren besteht letztendlich darin, dem ausführenden System eine Befehlssequenz mitzuteilen. Zur Programmlaufzeit wird diese dann sequenziell abgearbeitet.

Ein Befehl kann beispielsweise eine einfache Zuweisung sein:

```
1 int a = 1; // Der Variable a des Typs int wird der Wert 1 zugewiesen
```

Dieser Befehl stellt eigentlich schon eine Abkürzung dar und besteht aus zwei Komponenten:

```
1 int a; // Eine neue Variable a des Typs int wird deklariert
2 a = 1; // Der Variable a wird der Wert 1 zugewiesen
```

Jeder Befehl in Objective-C muss mit einem Semikolon enden.

### 3.1.1 Primitive Datentypen

Die wichtigsten primitiven Datentypen sind:

**int** 'integer': Ganze Zahl, z.B. `int a = -42;`  
**uint** 'unsigned integer': Nichtnegative ganze Zahl, z.B. `uint a = 1;`  
**char** 'character': Einzelnes Zeichen, z.B. `char a = 'x';`  
**float** 'floating-point number': Dezimalzahl, z.B. `float a = 3.14;`  
**double** 'double precision floating-point': Sehr lange Dezimalzahl  
**BOOL** 'boolean': Entweder wahr (**YES**) oder falsch (**NO**), z.B `BOOL a = YES;`

Die primitiven Datentypen in Objective-C und C sind äquivalent und ineinander überführbar. Deswegen kann beispielsweise statt **BOOL** auch **bool** oder Boolean verwendet werden. Die korrekte Objective-C Syntax ist aber **BOOL**.

### 3.1.2 Kommentare

In Objective-C wird Text, der wie im obigen Beispiel in einer einzelnen Zeile hinter zwei Schrägstrichen steht, als Kommentar aufgefasst und nicht ausgeführt. Mehzeilige Kommentare beginnen mit der Zeichenfolge `/*` und enden mit `*/`:

```
1 /* Der folgende Befehl
2 gibt den Text "Hello World"
3 als Output in der Konsole aus */
4 NSLog(@"%@", @"Hello World");
5 // Ausgabe: Hello World
```

Kommentare sind sehr wichtig, um Code lesbarer und sowohl für andere als auch für sich selbst längerer Zeit nachvollziehbar zu gestalten.

Verwendet in den Übungen häufig Kommentare!

### 3.1.3 Output

Der Befehl `NSLog(@"Text");`; wie im obigen Beispiel ist eigentlich C-Syntax, ist aber sehr nützlich um beliebigen Text in der Konsole ausgeben zu lassen. Dies dient hauptsächlich dazu, Ausgaben an bestimmten Stellen im Programmablauf zu platzieren um diesen nachvollziehen zu können und Fehler zu finden (Debugging).

### 3.1.4 Einfache Operationen

Die einfachen Rechenoperationen `+`, `-`, `*`, `/` und `%` (Modulo) sind direkt in Objective-C verfügbar.

Wichtig ist, zu beachten, dass zwischen integer- und floating-point-Division unterschieden wird. Nur wenn mindestens einer der Operanden ein floating-point Wert ist, wird auch ein solcher zurückgegeben, sonst wird abgerundet. Dies ist eine häufige Fehlerquelle!

Typen können mit der sog. `cast` Operation ineinander umgewandelt werden, bei der der Wert als der Typ interpretiert wird, der in Klammern davor angegeben wird: Mit `int a = (int)3.14;` hat `a` nun den Wert 3.

```

1 int a = (1 + 2) * 3; // a ist jetzt 9
2 float b = a / 2 // b ist jetzt 4, da sowohl a als auch 2 integer sind
3 float c; // Die neue Variable c wird als floating-point deklariert
4 /* Alle folgenden Operationen sind äquivalent und
5 setzen c auf 4.5, da floating-point Division verwendet wird */
6 c = a / 2.0
7 c = a / 2.
8 c = a / 2f
9 c = a / (float)2

```

Für die häufig verwendeten Operationen 'um 1 erhöhen/verringern' gibt es die Abkürzungen `a++`; bzw. `a--`:

```

1 int a = 0;
2 // Folgende Operationen sind äquivalent
3 a = a + 1;
4 a++;
5 // und
6 a = a - 1;
7 a--;

```

### 3.1.5 Abfragen

Grundlegend für die Programmierung sind weiterhin einfache wenn-dann Abfragen, die in Objective-C folgende Syntax haben:

```

1 int year = 2013;
2 if (year < 2013) {
3     NSLog(@"Vergangenheit");
4 } else if (year == 2013) {
5     NSLog(@"Gegenwart");
6 } else if (year > 2013) {
7     NSLog(@"Zukunft");
8 } else {
9     NSLog(@"unmöglich!");
10 }

```

Der Code in geschweiften Klammern wird also nur ausgeführt, wenn die Bedingung im if-Statement wahr (**YES**) ist.

Logische Operatoren:

**a == b** gleich

**a != b** ungleich

**a > b** größer, entsprechend auch **>=, <, <=**

**!a** logische Negation

**a && b** logisches Und

**a || b** logisches Oder

#### 3.1.6 Schleifen

Zusätzlich zu Abfragen bilden Schleifen einen weiteren Grundbaustein der Programmierung, um den gleichen Codeblock häufig hintereinander auszuführen.

**for**-Schleifen enthalten drei Segmente im Argument und einen Codeblock:

```
1 int a = 1;
2 for (int i=0; i<10; i++) {
3     a = a * (i+1);
4 }
5 // a ist jetzt 10! (Fakultät)
```

Das erste Segment **int i=1** (Initialisierung) wird zu Beginn der Schleife ausgeführt und initialisiert hier die Variable **int i** mit dem Wert 1. Anschließend wird die Schleife beendet, wenn der boolsche Ausdruck im zweiten Segment (Test) falsch ist, also **NO** zurückgibt. Andernfalls wird der Codeblock zwischen den geschweiften Klammern ausgeführt. Dann wird das dritte Segment (Fortsetzung) ausgeführt, das hier die Variable **i** um eins erhöht. Nun wird wieder der Ausdruck im zweiten Segment geprüft.

**while**-Schleifen enthalten nur ein Segment und einen Codeblock:

```
1 int a = 1;
2 while (a < 100) {
3     a = a * 2;
4 }
5 // a ist jetzt 128
```

Die Schleife wird beendet, wenn der boolsche Ausdruck im Argument falsch ist, andernfalls wird der Codeblock zwischen den geschweiften Klammern ausgeführt und wieder der boolsche Ausdruck geprüft.

In beiden Schleifen kann mit dem Befehl `continue` der restliche Code des aktuellen Durchlaufs übersprungen werden. Der Befehl `break` bricht die Schleife ab.

#### 3.1.7 Strings

Im `NSLog(@"Hello World!");` Befehl sind uns schon Zeichenfolgen, sog. **Strings**, begegnet. Ein String kann genau wie ein integer oder floating-point einer Variablen zugewiesen werden. Der zugehörige Typ ist `NSString`.

Der Präfix `NS` wird uns noch häufiger begegnen, da er vom Namen der Programmiersprache für das Betriebssystem NeXTStep stammt, aus dem sich später Mac OS X entwickelt hat. Viele grundlegende Komponenten von Objective-C beginnen mit diesem Präfix.

Strings sind keine primitiven Datentypen mehr wie z.B. `int`, sondern Objekte, die wir im Folgenden noch genauer kennenlernen werden. Variablen, die auf Objekte verweisen, müssen bei der Deklaration mit einem \* vor dem Variablenamen gekennzeichnet werden.

Eine Zeichenfolge wie `@"text"` erzeugt immer einen neuen String, der im Allgemeinen nicht veränderlich ist ('immutable'). Einer Variable kann aber natürlich zu einem späteren Zeitpunkt ein anderer String zugewiesen werden.

```
1 NSString *einName = @"Alice";
2 NSLog(einName); // Ausgabe: Alice
3 einName = @"Bob";
4 NSLog(einName); // Ausgabe: Bob
```

#### 3.1.8 String Formatierung

In Objective-C können Werte aus Variablen verschiedenen Typs in einen String eingefügt werden, indem ein Platzhalter verwendet wird, der dann mit dem entsprechenden Wert gefüllt wird. Die wichtigsten Platzhalter sind:

`%i` für `int` und `uint`

`%f` für `float`, die Anzahl der Dezimalstellen kann dabei mit `%.3f` für beispielsweise 3 Dezimalstellen spezifiziert werden.

`%d` für `double` mit der gleichen Syntax für Dezimalstellen wie bei `float`

`%c` für `char`

`%@` für Strings

Platzhalter werden in der Reihenfolge, wie sie im String auftauchen, mit dem Wert der folgendenden, kommagetrennten Variablen ersetzt:

```
1 int a = 1;
2 NSLog(@"%@", a); // Ausgabe: a hat den Wert 1
```

## 3.2 Grundlagen der objektorientierten Programmierung

Bei der Softwareentwicklung ist es nicht nur wichtig, Programme zu schreiben, die funktionieren und die richtigen Ergebnisse liefern, sondern auch den Programmcode sinnvoll zu strukturieren. Sobald ein Softwareprojekt wächst und nicht mehr nur einfache Skripte darstellt, sondern Komponenten wie Benutzeroberflächen und Datenstrukturen enthält, wird es schnell unübersichtlich und damit fehleranfällig. Projekte umfassen schnell einige Tausend Zeilen Code und sollten trotzdem einfach zu ergänzen und zu erweitern sein.

Bei der objektorientierten Programmierung strukturieren wir den Code in sinnvolle Einzelteile, die jeweils ein bestimmtes Element in unserem Programm repräsentieren. Dazu gehören bspw. Datenstrukturen oder auch Interfaceelemente wie Buttons und Labels.

### 3.2.1 Klassen & Objekte

Arbeiten wir also beispielsweise an einer Software, die Informationen zu verschiedenen Personen beinhaltet. Wir erstellen eine sog. **Klasse**, eine abstrakte Beschreibung oder 'Bauplan' einer Person. In unserem Programm können dann beliebig viele Instanzen dieser Klasse, oder **Objekte**, erzeugt werden, die nach diesem Bauplan erstellt wurden, aber voneinander unabhängig sind und jeweils eine bestimmte Person repräsentieren.

Wir können auch auf Basis einer existierenden Klasse eine sog. **Subklasse** definieren, die oft einen Spezialfall oder bestimmte Ausprägung dieser Klasse darstellt. Nach dem Prinzip der **Vererbung** erbt die Subklasse den 'Bauplan' ihrer Superklasse und kann diesen überschreiben oder erweitern.

Betrachten wir nun beispielsweise folgende Klasse:

#### Person.h

```

1 @interface Person : NSObject
2
3 // Attribute
4 @property (strong, nonatomic) NSString *name;
5
6 // Methoden
7 - (void)sayHi;
8
9 @end

```

#### Person.m

```

1 #import "Person.h"
2
3 @interface Person ()
4 // Privates Interface
5 @end
6
7 @implementation Person
8
9 - (void)sayHi {
10     NSLog(@"Hi, my name is %@", self.name);
11 }
12
13 @end

```

### 3.2.2 Interface & Implementierung

Eine Klasse besteht aus ihrem öffentlichen Interface und ihrer Implementierung. Meist werden diese in getrennten Dateien geschrieben: Das Interface in der Header-Datei mit Endung .h und die Implementierung in der Main-Datei mit Endung .m.

Damit der Compiler, der nur die Main-Dateien parst, auch den Code der zugehörigen Header-Datei berücksichtigt, müssen diese mit dem Aufruf `#import "Klassenname.h"` zu Anfang der .m Datei eingebunden werden.

Das öffentliche Interface stellt die Schnittstelle zum restlichen Programm dar und beschreibt, welche **Attribute** und **Methoden** die Klasse enthält.

Im Codebeispiel oben geben wir zunächst an, dass die Klasse 'Person' eine Subklasse von `NSObject` ist. `NSObject` ist die Basisstruktur von Objective-C mit wichtigen Mechanismen zur Speicherverwaltung u.ä. Fast jede Klasse in Objective-C geht auf `NSObject` zurück und erbt damit diese Mechanismen.

Zusätzlich kann die Main-Datei noch ein privates Interface enthalten, auf das nur innerhalb der Klasse zugegriffen werden kann. Dieses schreiben wir in der Main-Datei vor der Implementierung.

### 3.2.3 Attribute

Die Eigenschaften von Objekten einer Klasse werden durch ihre **Attribute** oder **Properties** repräsentiert.

Im Codebeispiel oben (*s. S. 36, Abschnitt 3.2.1*) spezifizieren wir, dass eine Person immer eine Variable `name` des Typs `NSString` hat.

`strong` kennzeichnet hier eine Option zur Speicherverwaltung und bewirkt, dass der Wert der Variable im Speicher gehalten wird bis das Objekt diesen freigibt. Dagegen stellt `weak` nur eine schwache Referenz auf den Speicher dar und verhindert nicht dessen Freigabe.

Wir werden noch Übung in der richtigen Verwendung dieser Kennzeichnungen bekommen. Bei Properties mit primitiven Datentypen kann diese Kennzeichnung weggelassen werden.

Wenn nicht mit mehreren Threads gearbeitet wird, sollten Properties immer mit `nonatomic` gekennzeichnet werden. Andernfalls treten einige ressourcenintensive Mechanismen in Kraft, um sicherzustellen, dass nicht gleichzeitig geschrieben und gelesen wird. Threads sind gleichzeitig ausgeführte Befehlssequenzen um beispielsweise in einem Hintergrundprozess Daten herunterzuladen und gleichzeitig das UI mit dem Ladefortschritt zu aktualisieren. Mit diesen werden wir uns zu einem späteren Zeitpunkt befassen.

#### 3.2.4 Methoden

Die Interaktion von Objekten miteinander geschieht auf Basis von Nachrichten, die untereinander ausgetauscht werden. Ein Objekt `sender` sendet eine solche Nachricht, indem es eine sog. **Methode `doSomething`** eines Objekts `receiver` aufruft:

```
1 [receiver doSomething];
```

Methoden können entweder einfach wieder eine Befehlsfolge abarbeiten oder zusätzlich einen Rückgabewert zurückgeben, der an der entsprechenden Stelle weiterverwendet werden kann.

Eine Methode definieren wir im öffentlichen Interface einer Klasse. Damit geben wir die Möglichkeit an, solche Nachrichten zu empfangen:

```
1 - (void)doSomething;
```

Die Implementierung der Methode in der Main-Datei enthält dann den auszuführenden Code, wenn diese Methode aufgerufen wird. Innerhalb der Implementierung kann mit dem Symbol `self` auf die eigene Instanz der abstrakten Klasse zugegriffen werden, also auf das Objekt selbst, das die Methode ausführt.

```
1 - (void)doSomething {
2     // Auszuführender Code
3 }
```

Im Codebeispiel (s. S. 36, Abschnitt 3.2.1) geben wir bspw. an, dass die Methode `sayHi` keinen Rückgabewert hat: (`void`). Andernfalls wird stattdessen der Typ des Rückgabewerts angegeben, also z.B. (`int`) oder (`NSString*`).

Hat die Methode einen Rückgabewert, kann er natürlich auch einer Variablen zugewiesen werden:

```
1 int count = [countingObject countSomething];
```

Weiterhin können Methoden Parameter annehmen, die in ihrer Implementierung verwendet werden. Parameter werden mit ihrem Datentypen hinter einem Doppelpunkt angegeben:

```

1 // im Interface:
2 - (float)divideNumber:(float)number byDivisor:(float)divisor;

1 // in der Implementierung:
2 - (float)divideNumber:(float)number byDivisor:(float)divisor {
3     float result = number/divisor;
4     return result;
5 }
```

### 3.2.5 Instanz- und Klassenmethoden

Im Allgemeinen verwenden wir Methoden, um mit bestimmten Objekten zu interagieren, sog. **Instanzmethoden**. In deren Implementierung kann dann auch auf Attribute des entsprechenden Objekts zugegriffen werden. Instanzmethoden werden mit dem Zeichen – vor der Deklarierung gekennzeichnet:

```
1 - (void)doSomething;
```

Es gibt Situationen, in denen eine Methode keinen Bezug zu einem bestimmten Objekt der Klasse hat. Wenn bspw. nur allgemeine Informationen über die Klasse abgefragt werden, kann eine Methode stattdessen mit dem Zeichen + als **Klassenmethode** gekennzeichnet werden.

Die Syntax zum Aufruf von Klassenmethoden ist äquivalent zu Instanzmethoden und verwendet das Symbol der Klasse als Empfänger:

```
1 [ReceivingClass getSomething];
```

In Apple's Frameworks sind Klassenmethoden häufig implementiert, um schnell häufig verwendete Objekte zu erstellen. Die Klassenmethode liefert dann als Rückgabewert ein neues Objekt der Klasse:

```

1 NSArray *array = [NSArray arrayWithObjects:a, b, c, nil];
2 NSString *string = [NSString stringWithFormat:@"Name: %@", self.name]
```

### 3.2.6 Polymorphie

Wir haben gelernt, dass Subklassen ihre Attribute und Methoden von ihrer Superklasse erben. Das Prinzip der Polymorphie ermöglicht uns, die Implementierung der Methoden der Superklasse zu überschreiben (sog. **overriding**).

In der Subklasse kann die Methode einfach erneut implementiert werden. Wird sie dann von einem Objekt aufgerufen, wird diese Implementierung anstatt der der Superklasse ausgeführt.

Es kann innerhalb einer Klasse außerdem mit dem Symbol `super` auf die Superklasse zugegriffen werden. So können Methodenaufrufe an die Implementierung der Superklasse weitergeleitet werden.

In einer Subklasse der Klasse `Person` können wir also beispielsweise die Methode `sayHi` erneut implementieren. Wird die Methode aufgerufen, wird dann stattdessen diese Neuimplementierung ausgeführt. Soll beim Methodenaufruf einfach etwas zusätzlich ausgeführt werden, so können wir die Superklassenimplementierung aufrufen und den zusätzlichen Code anschließend ausführen:

```
1 - (void)sayHi { // Die Implementierung der Superklasse wird mit dieser
   // Implementierung überschrieben
2   [super sayHi]; // Hier wird die Superklassenimplementierung aufgerufen
3   // zusätzlicher Code hier
4 }
```

#### 3.2.7 Getter und Setter Methoden

Um auf Attribute einer Klasse zuzugreifen, werden wiederum Methoden verwendet.

Für jede Property wird automatisch eine **Getter**- und einer **Setter**-Methode generiert.

Die Getter-Methode ist einfach eine Methode mit gleichem Namen wie die Property, die den Wert derselben als Rückgabewert liefert, während die Setter-Methode dem großgeschriebenen Variablenamen ein `set` voranstellt und als Parameter den neuen Wert annimmt.

Sie sind somit definiert als:

```
1 // Getter
2 - (Type)variable;
3 // Setter
4 - (void)setVariable:(Type)value;
```

Die Verwendung erfolgt wie bei regulären Methoden:

```
1 // Getter
2 [self variable] // Rückgabewert: Wert der Variable
3 // Setter
4 [self setVariable:value]; // Setzt den Wert der Variable auf value
```

Äquivalent zum Methodenaufruf in eckigen Klammern ist die häufig verwendete **dot-Syntax**:

### 3 Objective-C

```
1 // Getter  
2 self.variable // äquivalent zu [self variable]  
3 // Setter  
4 self.variable = value; // äquivalent zu [self setVariable:value]
```

Zusätzlich wird automatisch für jede Property eine Instanzvariable mit dem gleichen Namen und einem vorangestellten Unterstrich `_` generiert. Diese kann innerhalb der Klasse ebenso verwendet werden, um auf ein Attribut zuzugreifen:

```
1 _variable // Gibt den Wert der Variable zurück  
2 _variable = value; // Setzt den Wert der Variable auf value
```

Anders als bei der Dot-Syntax werden in diesem Fall jedoch **nicht** die Getter- und Setter-Methoden aufgerufen.

Getter- und Setter-Methoden können wie jede andere Methode überschrieben werden. Wir können sie also in unserer Klasse selbst implementieren, sodass nicht die automatisch generierten Methoden sondern unsere eigene Implementierung verwendet wird. In dieser Implementierung können wir dann offensichtlich nicht die entsprechende Getter- oder Setter-Methode aufrufen, da dies in den meisten Fällen zu einer Endlosschleife führt. Daher wird hier auf die Instanzvariable zurückgegriffen.

```
1 // Getter  
2 - (Type)variable {  
3     // do something  
4     return _variable;  
5 }  
  
1 - (void)setVariable:(Type)value {  
2     _variable = value;  
3     // do something  
4 }
```

Im Allgemeinen gibt die Getter-Methode den Wert der Variable zurück, während die Setter-Methode diesen entsprechend des Arguments der Methode setzt.

Wir können diese Methoden jedoch auch verwenden, um beliebige Werte zurückzugeben oder zu setzen. Häufig werden in der Getter-Methode bspw. auf Anfrage Objekte instantiiert:

```
1 - (Type)variable {  
2     if (!_variable) _variable = [[Type alloc] init];  
3     return _variable;  
4 }
```

### 3.2.8 Instanzierung von Objekten

An beliebigen Stellen im Code kann ein neues Objekt einer Klasse erstellt werden, also eine neue Instanz nach dem 'Bauplan' der Klasse:

```
1 Person *alice = [[Person alloc] init];
```

[`Person alloc`] ist ein Methodenaufruf, der Speicher für ein neues Objekt der Klasse `Person` bereitstellt. Anschließend wird das Objekt mit `init` initialisiert und der Variable `alice` vom Typ `Person` zugeordnet.

### 3.2.9 Verfügbarkeit von Klassen

Es kann nur auf Klassen zugegriffen werden, wenn diese dem Compiler an der entsprechenden Stelle bekannt sind. Am Anfang der entsprechenden Datei muss das Klasseninterface also mit dem Aufruf `#import "Klassenname.h"` verfügbar gemacht werden. Das gilt auch für die Main-Datei der gleichen Klasse!

#### Forward Declarations

Der Aufruf `#import "Klassenname.h"` bindet im Prinzip den Code der angegebenen Datei direkt an der entsprechenden Stelle ein, sodass dieser dem Compiler zur Verfügung steht. Dieser Aufruf kann jedoch zu einer Endlosschleife führen, wenn in der eingebundenen Datei wiederum die Einbindende importiert wird.

In solchen Situationen, in denen zwei Klassen jeweils voneinander abhängig sind, wird eine sog. **Forward Declaration** verwendet. Dabei wird die zu benötigte Klasse im Header zunächst nur mit dem Aufruf `@class Klassenname;` als verfügbar gekennzeichnet und dann erst in der Main-Datei importiert.

#### A.h

```
1 @class B;
2
3 @interface A : NSObject
4
5 @property (strong, nonatomic) B *b;
6
7 @end
```

#### A.m

```
1 #import "B.h"
2
3 @implementation A
4
5 @end
```

Es gilt außerdem die Konvention, Klassen nur in Main-Dateien zu importieren und in Headers stattdessen die Forward Declaration zu verwenden.

### 3.3 Symbolnamen & Konventionen

Symbolnamen müssen in Objective-C bestimmte Bedingungen erfüllen:

- Namen beginnen immer mit einem Buchstaben oder Unterstrich \_
- Darauf folgt eine beliebige Kombination aus Buchstaben, Unterstrichen und Ziffern.
- Einige Symbole sind reserviert und können nicht verwendet werden.

Außerdem sollten unbedingt die folgenden Konventionen eingehalten werden. Achtet bitte darauf diese Konventionen beim Schreiben von Objective-C Code zu verwenden, auch wenn ihr in anderen Programmiersprachen anders vorgeht. So kann bspw. die Autovervollständigung zuverlässig verwendet werden und die Lesbarkeit des Codes steigt.

- Variablen- und Methodennamen beginnen mit einem Kleinbuchstaben, Klassennamen mit einem Großbuchstaben
- Zur Repräsentation mehrerer Wörter werden keine Unterstriche verwendet. Stattdessen wird der Anfangsbuchstabe jedes folgenden Worts großgeschrieben (camel case). Beispiele:

- isEnabled
  - objectForIndex:
  - viewDidLoad

- Ein Methodename beginnt mit einer Beschreibung des Rückgabewerts oder der ausführenden Aktion und bezieht ihre Argumente in den Namen ein. Beispiele:

- filteredArrayUsingPredicate:
  - pushViewController:animated:

- Verwendet einen Klassenprefix für größere Projekte, sodass jede Klasse eindeutig benannt ist, denn es gibt keine Namespaces in Objective-C. Verwendet 2-3 Großbuchstaben, die eurem Company Identifier und/oder Product Name ähneln und beginnt jede Klasse diesem Prefix. Andernfalls besteht die Gefahr, dass Dateien gleichen Namens aus anderen Projekten zu Konflikten führen. Allen von Apple zur Verfügung

gestellten Dateien steht ein 'NS' (für NextStep, die Programmiersprache des Betriebssystems aus dem das heutige Mac OS X hervorging) oder 'UI' (für User Interface, hauptsächlich bei iOS-relevanten Dateien) voran. Beispiele:

- `UIView`
- `NSArray`

- Methodennamen besitzen kein Prefix und müssen nur innerhalb der Klasse eindeutig sein. In verschiedenen Klassen wird sogar häufig der gleiche Methodenname verwendet, wenn diese Methoden einen ähnlichen Sinn erfüllen. So überschreiben bspw. viele Klassen die Methode `description`, deren Rückgabewert vom Typ `NSString` eine Textrepräsentation des Objekts darstellen soll.

## 3.4 Einige wichtige Klassen

In Objective-C sind viele Grundelemente der Programmierung Objekte. Dazu gehören die folgenden häufig verwendet Datenstrukturen. Diese sind ausführlich in der Dokumentation beschrieben (s. S. 24, Abschnitt 2.4).

### NSString

Objekte der Klasse `NSString` repräsentieren Zeichenfolgen und können schnell mit der abkürzenden Syntax `@"text"` erstellt werden.

Häufig verwenden wir die Klassenmethode `stringWithFormat:`, um Werte von Variablen nach der Syntax der String Formatierung einzubinden (s. S. 35, Abschnitt 3.1.8).

```
1 NSString *string = @"text";
```

### NSNumber

`NSNumber` kann als Alternative zu primitiven Datentypen zur Repräsentation von Zahlen verwendet werden und bietet die notwendigen Mechanismen, um numerische Datentypen als Objekte zu behandeln und weiterzuverwenden. Aus den meisten numerischen Datentypen (so auch `BOOL`) kann mit der Syntax `@(x)` ein `NSNumber`-Objekt erstellt werden.

```
1 NSNumber *a = @(3.14);
2 NSNumber *b = @YES;
```

## NSArray

`NSArray` stellt eine geordnete Liste von Objekten dar und bietet viele Mechanismen, mit diesen Objekten zu arbeiten. Die abkürzende Syntax `@[a, b, c]` erstellt ein `NSArray` mit den entsprechenden Objekten in der angegebenen Reihenfolge.

Die Instanzmethode `objectAtIndex:` bietet dann Zugriff auf das Objekt mit dem entsprechenden Index in der Liste.

Um Listen schnell durchzugehen, existiert die erweiterte **Fast Enumeration** Syntax der `for`-Schleife in Objective-C:

```
1 NSArray *list = @[a, b, c];
2
3 for (NS0bject *object in list) {
4     // do something
5 }
```

Listen sind zunächst nicht veränderbar, es können also keine Objekte hinzugefügt oder entfernt werden. Stattdessen wird die Subklasse `NSMutableArray` verwendet, um solche veränderbaren Listen darzustellen. Diese implementiert die Instanzmethoden  `addObject:` und  `removeObject:`, sowie zusätzliche Möglichkeiten zur Manipulation der Liste.

## NSDictionary

Analog zu Listen repräsentieren Objekte der Klasse `NSDictionary` ungeordnete Key-Value Paare. Jedem Key des Typs `NSString` wird ein beliebiges Objekt (Value) zugeordnet. Abkürzend schreiben wir dann `@{@"key1":value1, @"key2":value2}`.

Auf ein bestimmtes Objekt kann mit der Instanzmethode `objectForKey:` zugegriffen werden.

Außerdem kann die gleiche Syntax zur Fast Enumeration verwendet werden wir bezüglich `NSArray`, wobei jedoch keine Reihenfolge der Aufzählung definiert ist.

```
1 NSDictionary *dic = @{@"key1":value1, @"key2":value2};
2
3 for (NS0bject *object in dic) {
4     // do something
5 }
```

Als veränderbare Subklasse steht hier das `NSMutableDictionary` zur Verfügung, das die Instanzmethoden  `setObject:forKey` und  `removeObjectForKey` implementiert.

# 4 iOS App Architektur

## 4.1 iOS App Lifecycle

Wir versuchen nun im Detail zu verstehen, wie unsere Apps auf einem iOS Gerät ausgeführt werden. Außerdem sind in einem Xcode Projekt verschiedene Dateien zu finden, die wir hier in Kontext bringen.

Die folgenden Informationen sind im iOS App Programming Guide [1] zu finden und können dort vertieft werden.

### 4.1.1 App States

Eine App liegt immer in einem der folgenden **App States** vor:

**Not running** Die App läuft nicht.

**Inactive** Ein Zwischenzustand, in dem die App im Vordergrund läuft aber keine Events empfängt, bspw. während des Startvorgangs oder bei Unterbrechungen wie eingehenden Anrufen.

**Active** Die App läuft normal im Vordergrund und empfängt Events.

**Background** Die App führt Code im Hintergrund aus. Wenn nicht explizit eine begrenzte Laufzeit zur Ausführung eines Hintergrundprozesses angefordert wird, geht die App direkt zum folgenden State über.

**Suspended** Die App befindet sich im Hintergrund und führt keinen Code aus. Es kann schnell wieder ein Vordergrund-State angenommen werden, da der Speicher noch nicht freigegeben wurde. Das System kann dies jedoch jederzeit tun.

---

<sup>1</sup><https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphonesosprogrammingguide/>

### 4.1.2 Startprozess einer iOS App

1. Der Benutzer drückt auf das App Icon oder startet die App auf eine andere Weise.
2. Die Methode `main()` in der Datei 'main.m' wird aufgerufen. Diese sieht typischerweise wie folgt aus und braucht selten verändert zu werden:

```

1 int main(int argc, char * argv[])
2 {
3     @autoreleasepool {
4         return UIApplicationMain(argc, argv, nil, NSStringFromClass([
5             AppDelegate class]));
6     }

```

Die `main()` Methode wird auch als **Entry Point** bezeichnet und entspricht ihrem Äquivalent in C und C++.

3. `main()` ruft `UIApplicationMain()` auf. Mit einem -Klick auf den Methodennamen können wir die Dokumentation dieser Methode in Apple's **UIKit** Framework anschauen (s. S. 47, Abb. 4.1).

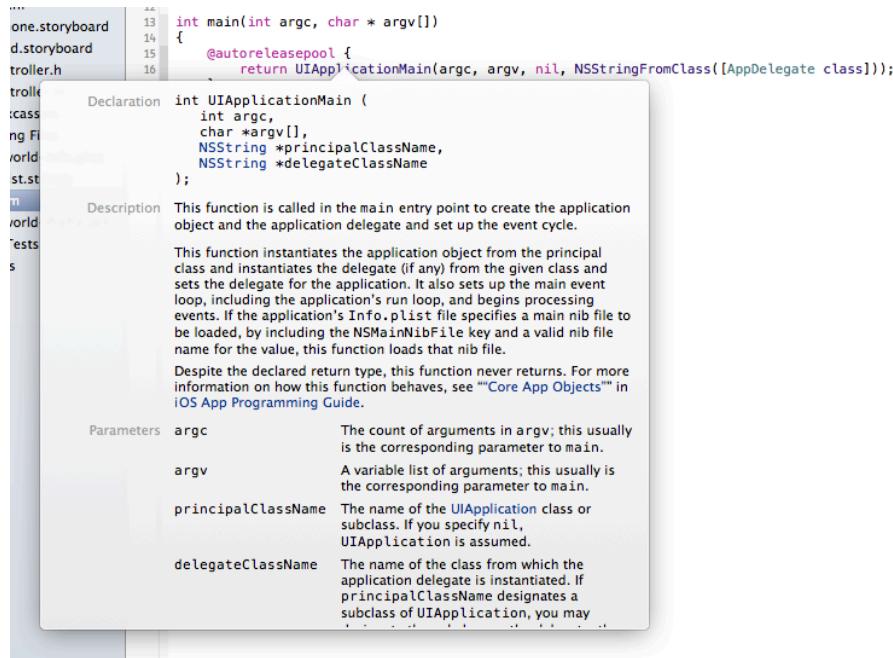


Abbildung 4.1: Die integrierte Dokumentation ist hilfreich, um den App Lifecycle zu verstehen

4. Diese Methode erzeugt das **Application Object**, also ein Objekt des Typs `UIApplication`, das für die zentrale Koordination der App verantwortlich ist.

U.a. stellt dieses Objekt die Verbindung zum Bildschirm her, erzeugt ein `UIWindow` zur Darstellung der Benutzeroberfläche und empfängt Benutzereingaben. Wir müssen für diese Klasse keinen weiteren Code schreiben, damit die Architektur der App funktioniert. Über den Aufruf einer Klassenmethode kann jederzeit auf das `UIApplication` Objekt zugegriffen werden:

1 `[UIApplication sharedApplication] // gibt das Application Objekt zurück`

5. Anschließend wird das **Application Delegate** erstellt.

Dabei wird die Klasse verwendet, die der `UIApplicationMain()` Methode in Form eines `NSString` Objekts übergeben wird:

1 `NSStringFromClass([XYZAppDelegate class]) // entspricht dem NSString @"XYZAppDelegate"`

Das erstellte Objekt dieser Klasse wird der Property `delegate` des Application Objects zugewiesen und ist daher immer verfügbar:

1 `[UIApplication sharedApplication].delegate // gibt das App Delegate Objekt zurück`

Wir implementieren die Klasse `XYZAppDelegate` wie jede andere Klasse in unserem Code mit ihrer Header- und Main-Datei.

Im Verlauf des Startvorgangs und während der Laufzeit der App werden bestimmte Methoden des Application Delegates aufgerufen. Diese informieren bspw. über den Wechsel der oben beschriebenen States und sind im Übersichtsdiagramm dargestellt (s. S. 49, Abb. 4.2).

6. Im Xcode Projekt ist eine Datei 'Info.plist' (meist mit vorangestelltem Product Name) zu finden. Diese steht dem Application Objekt zur Verfügung und enthält eine Liste verschiedener Optionen, die wir in der Projekt- und Targetkonfiguration anpassen können. Hier wird u.a. angegeben, wenn es ein **Storyboard** gibt, das als Startpunkt für die Darstellung der Benutzeroberfläche verwendet werden soll.

Dieses Storyboard wird geladen und dessen Initial Scene angezeigt. Alle im Storyboard enthaltenen Objekte werden dabei instanziert und die Connections (IBOutlets und IBActions) hergestellt.

7. Nun übernehmen unsere eigenen Klassen die Kontrolle. Unsere Implementierung des App Delegates wird hier häufig verwendet, um benötigte Datenstrukturen zu initialisieren. Außerdem ist das App Delegate verantwortlich dafür, auf die Veränderung der App States zu reagieren und bspw. rechenintensive Prozesse der Benutzeroberfläche (Animationen u.ä.) zu pausieren, wenn die App in den Hintergrund tritt.

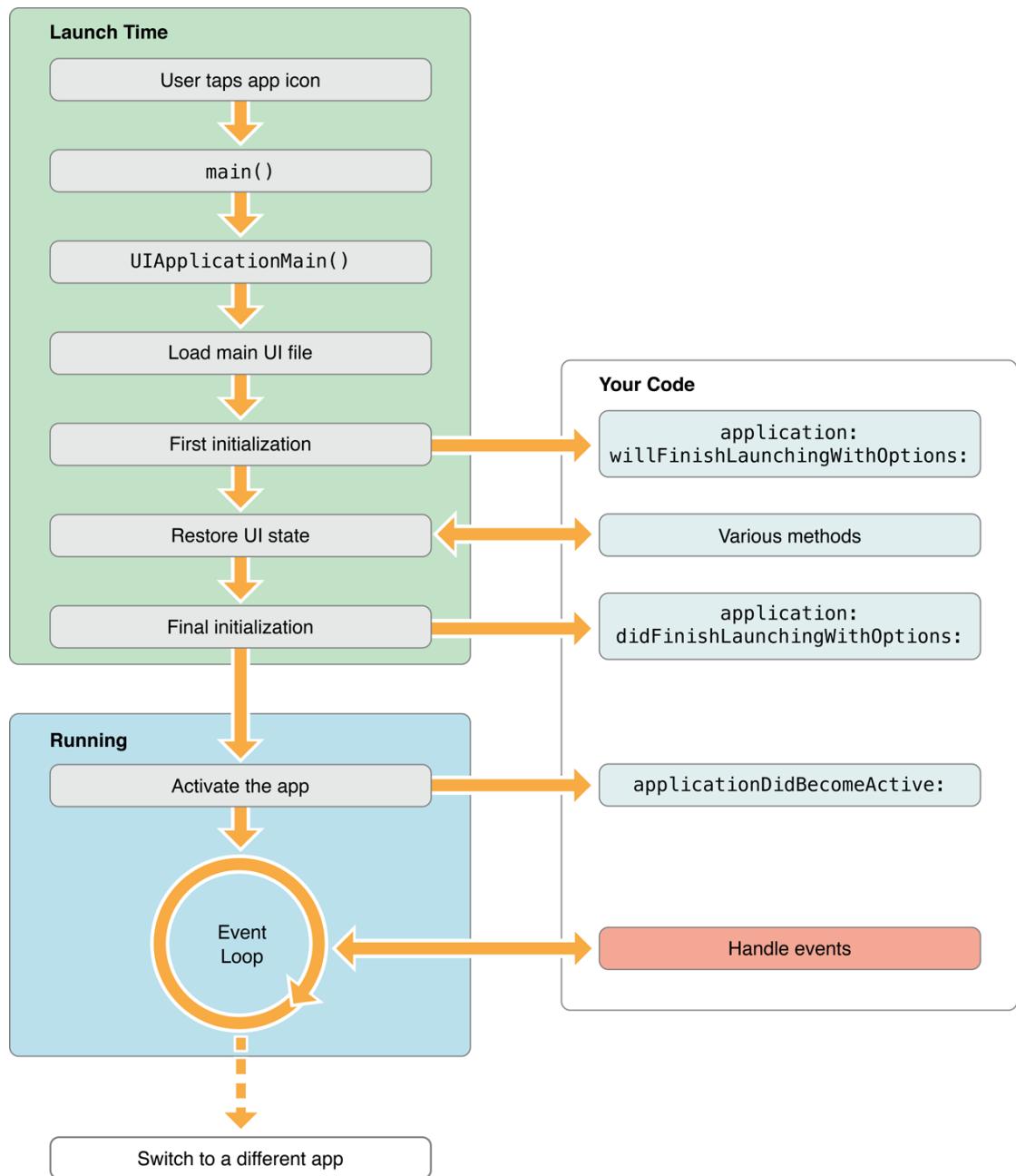


Abbildung 4.2: Übersicht der Prozesse beim Starten einer iOS App (aus dem iOS App Programming Guide)

## 4.2 Das Model-View-Controller Konzept

Für die Strukturierung komplexerer Apps müssen wir nun zunächst ein wichtiges Konzept der Softwareentwicklung verstehen:

Das **Model-View-Controller (MVC) Konzept** zieht sich konsequent durch die Gestaltungsmuster von iOS Apps und wird auch auf vielen anderen Plattformen verwendet.

Es basiert auf dem Grundsatz, dass **Modell**, **Präsentation** und **Steuerung** strikt getrennt implementiert werden (s. S. 50, Abb. 4.3). Durch diese Modularisierung bleibt ein Softwareprojekt flexibel und erweiterbar und einzelne Komponenten können leicht wiederverwertet werden.

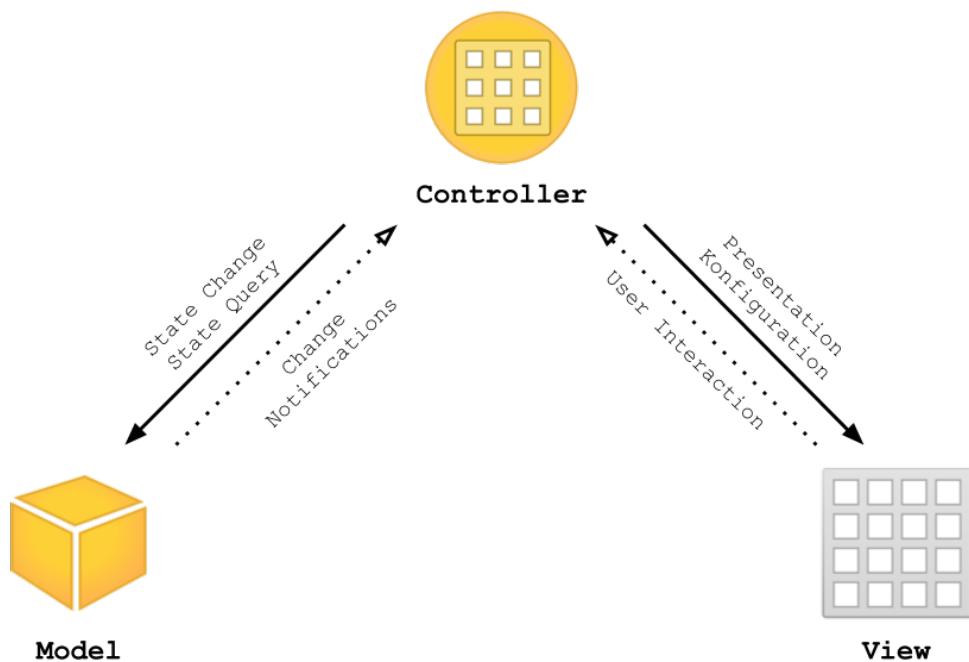


Abbildung 4.3: Das MVC Konzept trennt Modell, Präsentation und Steuerung

**Modell (Model)** Datenstrukturen und Logik werden dem Modell zugeordnet. Es stellt die darzustellenden Daten zur Verfügung und verarbeitet Anfragen bezüglich deren Modifikation.

Wir werden häufig Subklassen von `NSObject` mit Eigenschaften und Klassenbeziehungen implementieren, um solche abstrakten Datenstrukturen zu repräsentieren.

**Präsentation (View)** Subklassen von `UIView` repräsentieren Interfaceelemente und werden auf dem Bildschirm dargestellt. Sie werden vom Controller mit Informationen gefüllt und leiten Benutzereingaben an diesen weiter.

**Steuerung (Controller)** Der Controller verwaltet die Views und reagiert auf Benutzereingaben.

Zur Laufzeit der App übernehmen Subklassen von **UIViewController** die Kommunikation zwischen Model und View. Meist repräsentiert jeweils ein View Controller eine bestimmte Ansicht auf dem Bildschirm. In Reaktion auf Benutzereingaben stellt der View Controller Anfragen an das Model und konfiguriert die Views.

Zusätzlich sind übergeordnete Controller wie bspw. Instanzen von **UINavigationController** für die Koordination der einzelnen View Controller zuständig und verwalten deren hierarchische Strukturen.

## 4.3 View Hierarchie

Die **View** Komponente jeder iOS App ist für die Anzeige des User Interface auf dem Bildschirm verantwortlich. Wir verwenden dazu Subklassen von **UIView**.

**UIView** wird in Apples UIKit Framework als Subklasse von **UIResponder**: **NSObject** implementiert. Die Klasse erbt somit zusätzlich zu den Verwaltungsmechanismen von **NSObject** auch die Methoden zur Reaktion auf Benutzereingaben von **UIResponder**.

UIKit stellt viele Subklassen von **UIView** zur Verfügung, wie bspw. **UILabel**, **UIButton** und **UIImageView**. Diese implementieren jeweils Methoden, um ihren Inhalt auf dem Bildschirm zu rendern.

Jedes **UIView** Objekt präsentiert jedoch nicht nur seinen eigenen Inhalt sondern dient auch als Container für andere **UIView** Objekte. Somit erhalten wir eine hierarchische Struktur von **Superviews** mit jeweils beliebig vielen **Subviews** (s. S. 52, Abb. 4.4). An oberster Stelle der Hierarchie steht dabei ein Objekt der Klasse **UIWindow**: **UIView**, das von dem Application Object verwaltet wird (s. S. 47, Abschnitt 4.1.2).

Mit Instanzmethoden wie **addSubview:** und **removeFromSuperview** von **UIView** können wir der View Hierarchie Objekte hinzufügen oder Objekte entfernen.

### 4.3.1 Frame und CGRect

Jedes Objekt der Klasse **UIView** verwaltet einen Anzeigebereich, der durch das Attribut **CGRect frame** bestimmt ist. Der Frame bestimmt ein Rechteck im zweidimensionalen Koordinatensystem der Superview.

Der Ursprung des Koordinatensystems liegt dabei immer in der oberen linken Ecke der Superview (s. S. 52, Abb. 4.5) mit einer horizontalen x-Achse und vertikalen y-Achse in positiver Richtung nach rechts unten.

#### 4 iOS App Architektur

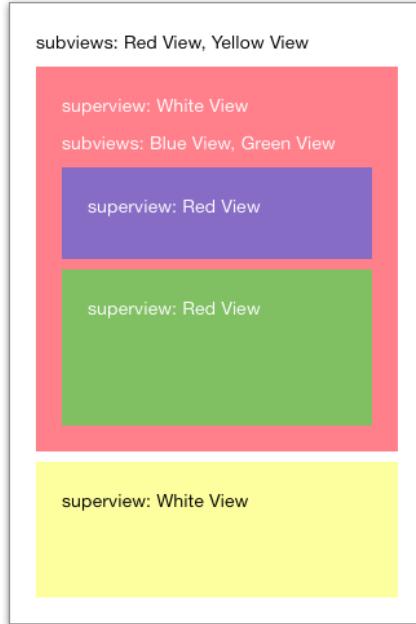


Abbildung 4.4: Jedes Objekt der `UIView` Klasse dient wieder als Container für weitere Objekte dieser Klasse



Abbildung 4.5: Der Ursprung des Koordinatensystems von `UIView` liegt in der oberen linken Ecke

`CGRect` ist keine Klasse sondern ein **Struct**, das ein Rechteck mit einem Ursprung `CGPoint` `origin` und einer Größe `CGSize` `size` repräsentiert. `CGPoint` stellt dabei einen Punkt im zweidimensionalen Koordinatensystem mit den Attributen `CGFloat` `x` und `CGFloat` `y` dar, während `CGSize` eine Ausdehnung mit Breite `CGFloat` `width` und Höhe `CGFloat` `height` beschreibt. `CGFloat` ist äquivalent zu `float` auf einer 32-bit Architektur und zu `double` auf einer 64-bit Architektur.

```
1 struct CGRect {  
2     CGPoint origin;  
3     CGSize size;  
4 };  
5  
6 struct CGPoint {  
7     CGFloat x;  
8     CGFloat y;
```

```

9  };
10
11 struct CGSize {
12     CGFloat width;
13     CGFloat height;
14 }

```

Zur Erstellung eines `CGRect` kann die Funktion `CGRectMake(x, y, width, height)` verwendet werden.

### 4.3.2 UIView Objekte

Wir können ein `UIView` Objekt u.a. mit der `initWithFrame:` Methode erstellen und dann der View Hierarchie hinzufügen, sodass es auf dem Bildschirm angezeigt wird:

```

1 UIView *view = [[UIView alloc] initWithFrame:CGRectMake(0, 0, 320, 44)];
2 [self.view addSubview:view]; // Angenommen self.view ist bereits Teil der View
    Hierarchie

```

Subklassen von `UIView` erben diesen Mechanismus. Ein `UILabel` lässt sich also bspw. genauso erzeugen und anzeigen.

## 4.4 Auto Layout

iOS Apps werden mittlerweile auf einer Vielzahl von Geräten unterschiedlicher Größe ausgeführt. Die Benutzeroberfläche unserer Apps sollte sich dabei dynamisch verschiedenen Anzeigegrößen und Inhalten anpassen können. Dazu gehören neben Displaygrößen und -orientierungen bspw. auch Sprachen mit verschiedenen Leserichtungen und Wortlängen.

Natürlich können wir versuchen, die Parameter des `frame` Attributs einer View in Reaktion auf Änderungen der Größe der Superview oder des Inhalts geschickt anzupassen. Häufig haben wir jedoch ganz bestimmte Vorstellungen, wie die Views einer View Hierarchie **zueinander** positioniert sein sollen. Bei der Konzeption von Benutzeroberflächen treten anstatt von festen Positionen vielmehr Regeln auf wie:

- Eine View soll wenn möglich zentriert positioniert sein, nie jedoch den Frame einer anderen View überlagern.
- Der Abstand einer View von beiden Seiten des Bildschirms soll 20pt betragen, während ihre Breite entsprechend flexibel ist.
- Zwei Views sollen immer die gleiche Höhe besitzen, die sich an der benötigten Größe der Subview Hierarchie der jeweils größeren View orientiert.
- Eine View soll immer den gesamten Bereich ihrer Superview ausfüllen.

Das **Auto Layout** Konzept der Objective-C Programmierung basiert auf der Definition von Regeln dieser Art, genannt **Constraints**.

#### 4.4.1 Constraints

Jedes Constraint repräsentiert eine Beziehung zwischen zwei **Attributen**  $x$  und  $y$  verschiedener Views, die durch den Ausdruck

$$y = m * x + b \quad (4.1)$$

gegeben ist. Dabei beschreiben  $m$  und  $b$  vom Typ **float Multiplier** und **Constant** der Beziehung.

In dieser Form können wir bspw. eine Constraint definieren, die den horizontalen Abstand zweier Views auf einen Wert von 20pt beschränkt:

```
1 secondView.left = firstView.right * 1.0 + 20.0
```

Die Auto Layout Syntax verfügt über die Attribute `left`, `right`, `top`, `bottom`, `leading`, `trailing`, `width`, `height`, `centerX`, `centerY` und `baseline`, wobei `leading` und `trailing` abhängig von der Leserichtung der eingestellten Sprache äquivalent oder umgekehrt zu `left` und `right` sind.

Es ist darüber hinaus möglich, neben Gleichheitsbeziehungen auch Ungleichungen zu definieren und Constraints ein Prioritätslevel zwischen 1 und 1000 zuzuweisen.

Zur Laufzeit ist die Superview für die automatische Positionierung ihrer Subviews entsprechend der definierten Constraints zuständig.

Für Views können beliebig viele, sich überlagernde Constraints definiert werden. Stehen einige Constraints in Konflikt zueinander und können nicht gleichzeitig erfüllt werden, hebt die Superview Constraints nacheinander auf, bis das Layout gültig ist. Dabei werden nicht-erfüllbare Constraints trotzdem so weit wie möglich einbezogen.

Enthält eine Superview keine Constraints, wird auf die expliziten Frames der Views zurückgegriffen.

Bei der Konstruktion der Constraints sollte unbedingt auf ein eindeutiges Layout geachtet werden.

#### 4.4.2 Intrinsic Content Size

Subklassen von **UIView** wie repräsentieren häufig Inhalt, dessen Darstellung eine bestimmte Größe erfordert. Dazu gehören bspw. **UILabel** oder **UIImageView** Objekte, deren Größe durch ihren Text- oder Bildinhalt bestimmt werden.

Diese **Intrinsic Content Size** wird verwendet, um die Größe einer View zu berechnen. In den meisten Fällen sollte die Intrinsic Content Size nicht durch explizite width und height Constraints beschränkt werden.

Die beiden Parameter **Content Hugging Priority** und **Compression Resistance Priority** einer View legen fest, mit welcher Priorität sich das Auto Layout System an der Intrinsic Content Size zu orientieren hat. Dabei führt eine geringe Content Hugging Priority bspw. dazu, dass eine View verfügbaren Platz über die Intrinsic Content Size hinaus eher einnimmt, als eine View mit höherer Content Hugging Priority. Die Compression Resistance Priority bestimmt hingegen, welche View zuerst auf eine geringere Größe als ihre Intrinsic Content Size gestaucht wird, wenn Constraints den verfügbaren Platz einschränken.

#### 4.4.3 Auto Layout im Interface Builder

Auto Layout kann für Interface Builder Dateien wie Storyboards einzeln aktiviert oder deaktiviert werden. Dazu ist im File Inspector eine Schaltfläche 'Use Autolayout' zu finden (s. S. 55, Abb. 4.6).

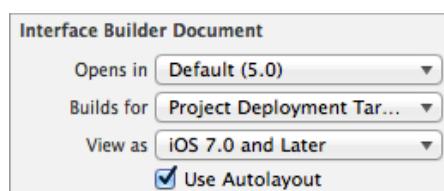


Abbildung 4.6: Auto Layout kann auf Basis einzelner Interface Builder Dateien ein- oder ausgeschaltet werden

Ist Auto Layout eingeschaltet, können wir Constraints auf verschiedene Weise erstellen. Eine Möglichkeit besteht in der Verwendung der Schaltflächen am unteren rechten Rand des Editorbereichs (s. S. 55, Abb. 4.7). Hier finden wir die interaktiven Menüs 'Align' und 'Pin', mit denen den ausgewählten Views Constraints hinzugefügt werden können.



Abbildung 4.7: Die Schaltflächen am unteren rechten Rand des Editorbereichs bieten Zugriff auf viele Auto Layout Optionen

Alternativ kann das von IBOutlets und IBActions bekannte Ziehen einer Verbindungsleitung bei gehaltener **ctrl**-Taste auch zum Erstellen von Constraints zwischen zwei Interfaceelementen verwendet werden (s. S. 56, Abb. 4.8). Dabei wird abhängig von der Richtung der

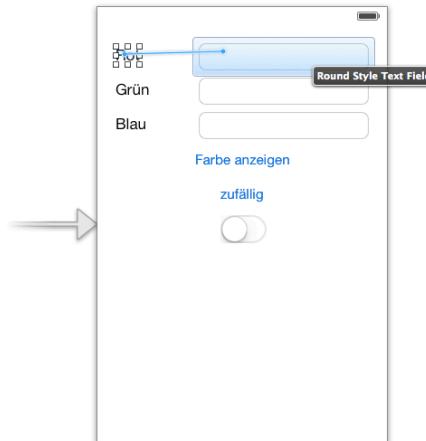


Abbildung 4.8: Constraints können ähnlich wie IBOutlets und IBActions durch Ziehen einer Verbindungsleitung erstellt werden

gezogenen Linie ein kontextabhängiges Menü gezeigt. Ziehen wir bspw. eine horizontale Linie, werden Constraint-Optionen bezüglich der Horizontalrichtung angezeigt.

Drei Farben markieren den Status der Constraints einer ausgewählten View im Interface Builder (s. S. 56, Abb. 4.9). Solange das Layout für die View noch uneindeutig ist, werden die Constraints in gelb angezeigt. Eindeutige Layouts werden blau und Layouts mit Konflikten rot markiert.

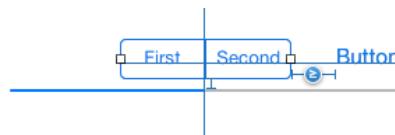


Abbildung 4.9: Ist das Layout einer View durch Constraints eindeutig festgelegt, werden diese blau gekennzeichnet

Wenn die Position der ausgewählten View im Interface Builder nicht ihrer berechneten Position gemäß ihren Constraints zur Laufzeit entspricht, wird sie diese gelb gestrichelt markiert (s. S. 57, Abb. 4.10). Im 'Resolve Auto Layout Issues' Menü am unteren rechten Bildschirmrand ist die Option 'Update Frames' zu finden, mit der die Position entsprechend angepasst werden kann.

Alle Constraints einer bestimmten View sind im Size Inspector aufgelistet. Eine Constraint kann wie jedes andere Objekt ausgewählt und mit dem Attributes Inspector bearbeitet werden. Die wichtigsten Optionen sind außerdem mit einem Doppelklick auf eine Constraint im Editorbereich erreichbar (s. S. 57, Abb. 4.11).



Abbildung 4.10: Uneindeutige Layouts oder Views, deren Position von vom berechneten Layout abweichen, werden gelb gekennzeichnet und zeigen ihre Position zur Laufzeit als eine gestrichelte Markierung

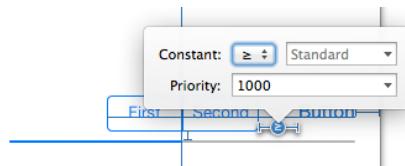


Abbildung 4.11: Ein Doppelklick auf eine Constraint im Interface Builder zeigt die wichtigsten Optionen in einem Popup

#### 4.4.4 Auto Layout im Code

Der Interface Builder stellt die effektivste Möglichkeit dar, ein eindeutiges und dynamisches Layout zu konzipieren. Im Code werden häufig nur die Konstanten einzelner Constraints zur Laufzeit angepasst und selten ganze Layouts konstruiert.

Trotzdem können wir Constraints als Objekte der `NSLayoutConstraint` Klasse im Code erstellen und einer View hinzufügen. Dazu implementiert `UIView` die Instanzmethoden `addConstraint:` und `removeConstraint:` und gewährt über das Attribut `NSArray*constraints` Zugriff auf alle Constraints.

Einzelne Constraints können mit der Klassenmethode `constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:` instanziert werden. Da jedoch meist mehrere Constraints benötigt werden, stellt `NSLayoutConstraint` zusätzlich eine Klassenmethode `constraintsWithVisualFormat:options:metrics:views:` zur Verfügung, die eine **Visual Format Language** verwendet.

Dabei übergeben wir der Klassenmethode einen String im ASCII-Art Stil, der die zu erstellenden Constraints beschreibt. So können wir bspw. einen Abstand von 10pt zwischen zwei Views `view1` und `view2` mit dem String

```
1 @"[view1]-10-[view2]"
```

darstellen. Sollen beide Views zusätzlich den Standardabstand von der Begrenzung durch die Superview besitzen, schreiben wir:

```
1 @"|[view1]-10-[view2]-|"
```

Die Syntax der Visual Format Language ist im Auto Layout Guide<sup>[2]</sup> einsehbar.

In dieser Form erstellte Constraint können wir dann einer Superview hinzufügen:

```
1 NSArray *constraints = [NSLayoutConstraint constraintsWithVisualFormat:@"|- [view1]-10-[view2]-|" options:0 metrics:nil views:  
    NSDictionaryOfVariableBindings(self.view1, self.view2)];  
2 [self.view addConstraints:constraints];
```

## 4.5 View Controller Hierarchy

Eine App besteht im Allgemeinen aus verschiedenen Bildschirmansichten, die jeweils eine Komponente der Benutzeroberfläche repräsentieren. Daher verwalten wir die View Hierarchy unserer App nicht zentral, bspw. im Application Delegate, sondern verwenden einzelne Klassen, die jeweils einen Teil der View Hierarchy verwalten.

Diese Subklassen von `UIViewController` sind der **Controller**-Komponente des Model-View-Controller Konzepts zugeordnet (s. S. 50, Abschnitt 4.2).

Jeder View Controller ist für die Verwaltung seiner eigenen **Content View** zuständig. `UIViewController` besitzt dafür ein Attribut `UIView*view`.

Zu den Aufgaben eines View Controllers gehören:

- die dynamische Positionierung der Views in der View Hierarchy seiner Content View
- die Kommunikation mit der Model-Komponente, um die Views mit Daten zu füllen
- die Reaktion auf Benutzereingaben

Nach dem Prinzip des **View Controller Containment** gibt es auch hier, ähnlich wie bei der View Komponente, eine hierarchische Struktur (s. S. 59, Abb. 4.12). Demnach gibt es übergeordnete View Controller, die die Content Views anderer View Controller der View Hierarchy ihrer eigenen Content View hinzufügen (s. S. 60, Abb. 4.13).

---

<sup>2</sup><https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/VisualFormatLanguage/VisualFormatLanguage.html>

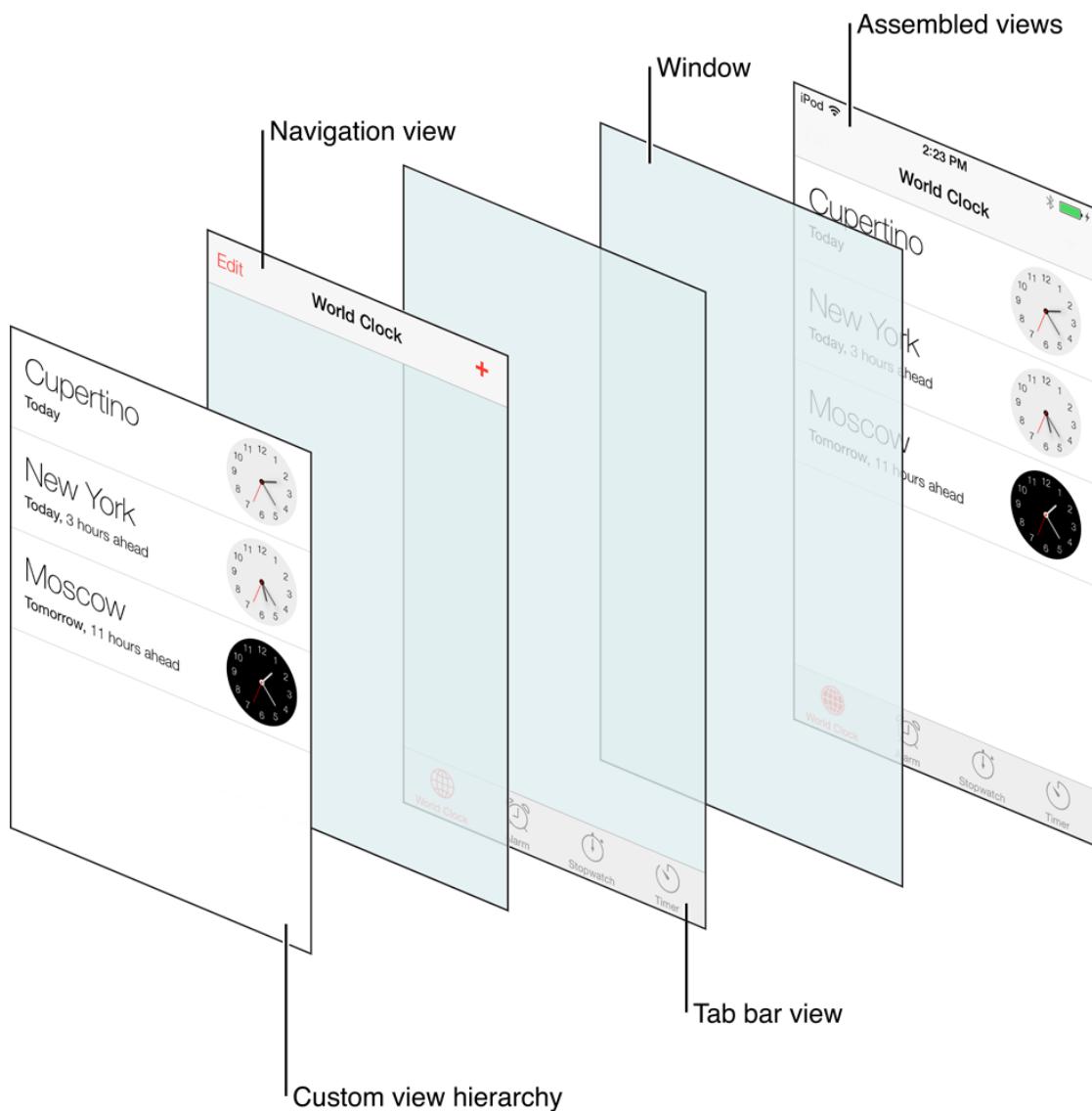


Abbildung 4.12: Container View Controller präsentieren Content Views anderer View Controller, sodass eine View Controller Hierarchy entsteht. Abbildung aus der UIViewController Class Reference

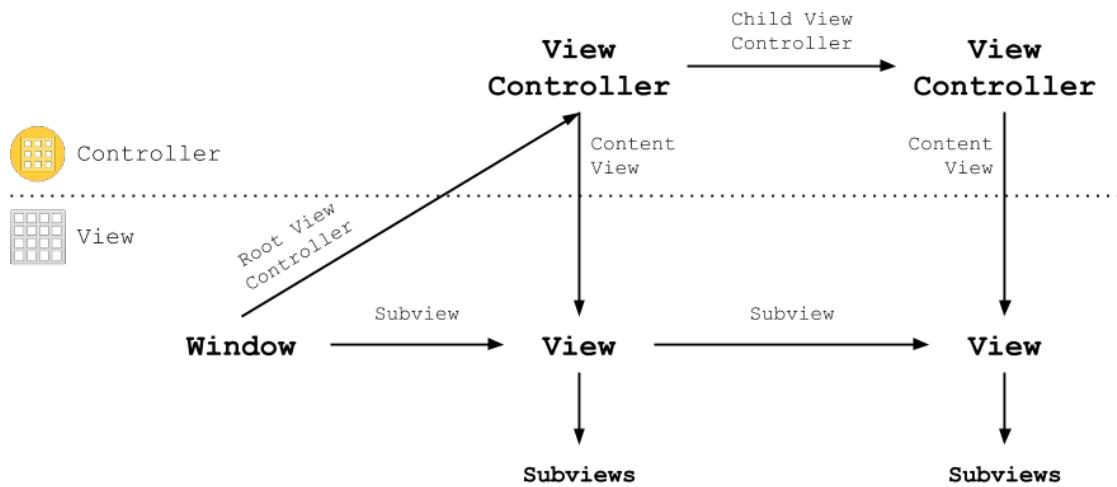


Abbildung 4.13: Jeder View Controller verwaltet seine Content View, die einen Teil der View Hierarchie darstellt

Die oberste Instanz der View Hierarchie ist ein Objekt der `UIWindow` Klasse. Analog besitzt `UIWindow` ein Attribut `UIViewController*rootViewController`, das die oberste Instanz der View Controller Hierarchie darstellt. Weisen wir diesem Attribut ein View Controller Objekt zu, so wird dessen Content View der View Hierarchie des `UIWindow` Objekts hinzugefügt. Wird ein Storyboard verwendet, geschieht dies automatisch mit der Option 'Initial View Controller' im Attributes Inspektor des entsprechenden View Controllers.

Während nur in wenigen Fällen Subklassen von `UIView` implementiert werden, wird man keine iOS App ohne mindestens eine Subklasse von `UIViewController` finden. Meist stellt die Implementierung der View Controller den Großteil der Programmierung von iOS Apps dar.

#### 4.5.1 View Controller Lifecycle

Im Verlauf der Präsentation eines View Controllers können verschiedene Instanzmethoden überschrieben werden, um auf Änderungen der Darstellung zu reagieren:

- `viewDidLoad` wird aufgerufen, sobald die Content View geladen wurde. An dieser Stelle können Attribute, die im Storyboard nicht hinreichend konfiguriert werden können, dynamisch angepasst werden, um einen Ausgangspunkt für die Präsentation zu schaffen.
- `viewWillAppear:`, `viewWillDisappear:`, `viewDidAppear:` und `viewDidDisappear:` können verwendet werden, um die Inhalte der Content View zu aktualisieren oder Vorgänge zu beginnen/anzuhalten.

- `didReceiveMemoryWarning` wird aufgerufen, wenn das System die App auffordert, Speicher freizugeben. Hier sollten nicht mehr benötigte Objekte oder solche, die leicht wieder zu erstellen sind, freigegeben werden.

### 4.5.2 Präsentation von View Controllern

Die `UIViewController` Klasse implementiert bereits eine einfache Möglichkeit, einen anderen View Controller temporär zu präsentieren. Die Instanzmethode `presentViewController:animated:completion:` fügt die Content View des entsprechenden View Controllers der View Hierarchie hinzu, während `dismissViewControllerAnimated:completion:` die Präsentation wieder beendet.

```
1 // Präsentiert einen View Controller
2 [self presentViewController:modalViewController animated:YES completion:nil];
3 // Beendet die Präsentation
4 [self dismissViewControllerAnimated:YES completion:nil];
```

### 4.5.3 Container View Controller in UIKit

Das UIKit Framework stellt einige nützliche Subklassen von `UIViewController` zu Verfügung, die in vielen Apps strukturgebend verwendet werden.

Insbesondere werden wir uns noch genauer mit dem äußerst vielseitigen `UITableViewController` befassen. Dieser gehört mit `UICollectionViewController` zu den inhaltsbasierten Subklassen von `UIViewController`.

`UINavigationController` und `UITabBarController` hingegen sind als Container View Controller konzipiert. Sie enthalten selbst nur wenige Subviews in Form von Leisten. Stattdessen verwalten sie eine Hierarchie von weiteren View Controllern und präsentieren deren Content Views.

#### Navigation Controller

`UINavigationController` implementiert eine **Stack**[<sup>3</sup>] Datenstruktur, die in Form eines Attributs `NSArray*viewControllers` nach außen repräsentiert wird.

Einem Attribut `UIViewController*rootViewController` kann zunächst ein View Controller zugewiesen werden, dessen Content View an erster Stelle der Hierarchie stehen soll. Anschließend kann dem Stack mit Aufrufen der Instanzmethode `pushViewController:animated:` ein View Controller hinzugefügt und `popViewControllerAnimated:` der oberste View Controller entfernt werden.

---

<sup>3</sup><http://de.wikipedia.org/wiki/Stapelspeicher>

Jeder von einem Navigation Controller verwaltete View Controller kann über das Attribut `UINavigationController*navigationController` auf diesen zugreifen.

```

1 // Präsentiere einen View Controller
2 [self.navigationController pushViewController:nextViewController animated:YES
   ];
3 // Zurück zum vorherigen View Controller
4 [self.navigationController popViewControllerAnimated:YES];

```

Die in den Subviews der Content View des Navigation Controllers präsentierten Elemente, bspw. der Titel und die Buttons in Navigationsleiste und Toolbar, sind von dem jeweils präsentierten View Controller abhängig. Jeder View Controller besitzt dafür ein Attribut `UINavigationItem*navigationItem`, das entsprechend konfiguriert werden kann und diese Informationen dem Navigation Controller bereitstellt.

```

1 self.navigationItem.title = @"Titel";
2 self.title = @"Titel"; // äquivalente Abkürzung
3
4 self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc]
   initWithBarButtonSystemItem:UIBarButtonSystemItemDone target:self action
  :@selector(doneButtonPressed:)];

```

#### 4.5.4 View Controller in Storyboards

Die beschriebenen Mechanismen der View Controller Hierarchie können und sollten zum Großteil in das verwendete Storyboard ausgelagert werden.

Ein Storyboard ist in **Scenes** strukturiert, die jeweils einen View Controller und seine Content View Hierarchie repräsentieren. Aus der Object Library können dem Storyboard View Controller hinzugefügt werden.

Wird ein View Controller ausgewählt, können wir dessen Identität anpassen und die entsprechende Subklasse von `UIViewController` auswählen, die wir implementiert haben (s. S. 62, Abb. 4.14). Zur Laufzeit wird der View Controller dann als Objekt dieser Subklasse instanziiert.

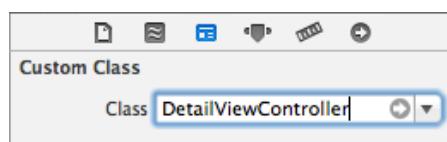


Abbildung 4.14: Im Identity Inspector kann bei ausgewählten View Controller dessen Subklasse ausgewählt werden

Anschließend stehen die im Code definierten IBOutlets und IBActions der `UIViewController` Subklasse im Interface Builder zur Verfügung und können mit Interfaceelementen verbunden werden.

## Segues

Zwischen Scenes vermitteln **Segues**. Diese stellen Beziehungen zwischen View Controllern dar und können zu deren Präsentation verwendet werden (s. S. 63, Abb. 4.15).

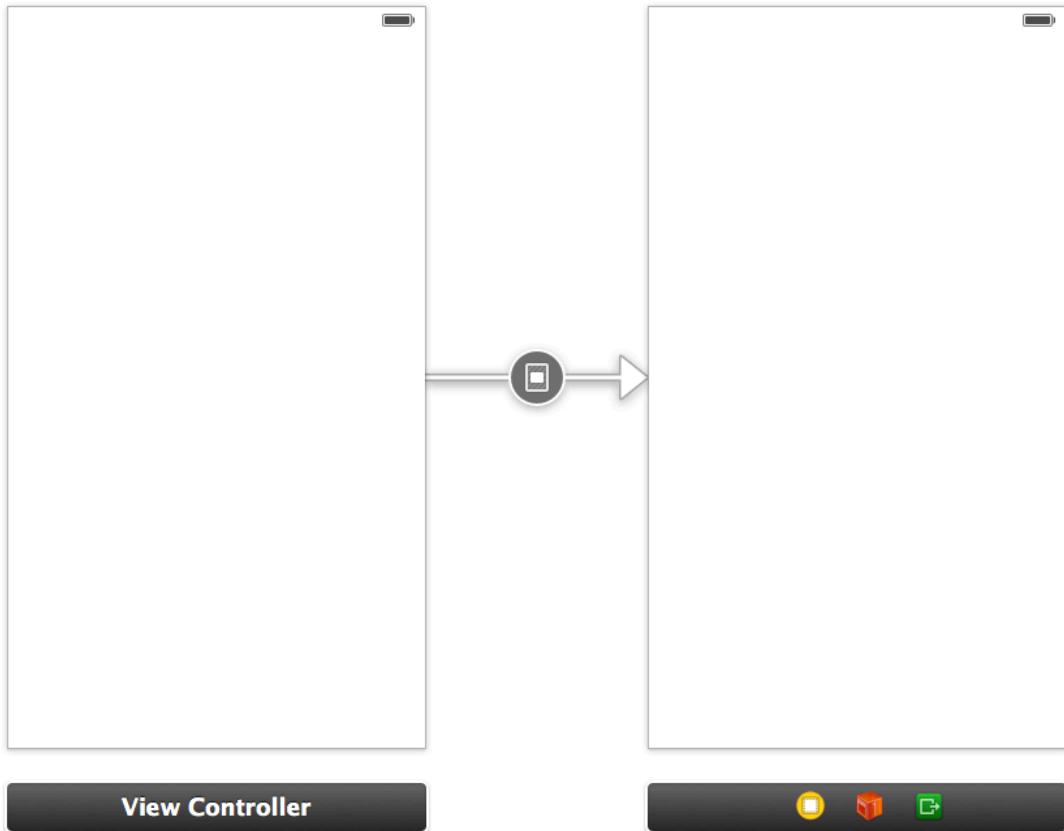


Abbildung 4.15: Segues vermitteln zwischen Scenes

Ähnlich wie bei IBOutlets und IBActions innerhalb einer Scene gibt es mehrere Möglichkeiten, Segues zu erstellen. Dazu gehören bspw. das Ziehen einer Verbindungsleitung mit gehaltener **ctrl**-Taste und die Verwendung des 'Triggered Segues' Abschnitts des Connection Inspectors.

Zwischen View Controllern kann es direkte Beziehungen geben, die durch eine **RelationshipSegue** gekennzeichnet werden. In dieser Form wird bspw. einem Navigation Controller sein Root View Controller zugewiesen.

Außerdem können Subklassen von **UIControl** wie bspw. **UIButton** bei einem **UIControlEvents** Segues auslösen. So lassen sich Übergänge zwischen View Controllern bereits innerhalb des Storyboards erstellen.

- **Modal Segues** sind das Äquivalent zum Aufruf der `presentViewController:animated:completion:` Methode.
- Befindet sich der View Controller des Ausgangspunkts der Segue in der View Hierarchie eines Navigation Controllers, kann eine **Push Segue** als Äquivalent zu der `pushViewController:animated:` Methode erstellt werden.

Im Attributes Inspector können Optionen bezüglich Übergangsanimation und Präsentationsmodus gewählt werden. Zusätzlich sollte hier ein Identifier vergeben werden.

Im Code können wir auf das Auslösen einer Segue reagieren, um den View Controller des Ziels entsprechend zu konfigurieren. Dazu überschreiben wir die `prepareForSegue:sender:` Instanzmethode und verwenden den Identifier:

```

1 - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
2     if ([segue.identifier isEqualToString:@"showDetailSegue"]) {
3         segue.destinationViewController.detailObject = detailObject;
4     }
5 }
```

## 4.6 Das Delegate Konzept

In der Programmierung komplexerer Apps ist es häufig notwendig, dass Klassen Informationen untereinander austauschen. Das **Delegate Konzept** ist ein einfacher und vielverwendeter Mechanismus der **einseitigen** Kommunikation zwischen Klassen, das einem einfachen **Frage-Antwort-Prinzip** folgt (s. S. 64, Abb. 4.16).



Abbildung 4.16: Das Delegate Konzept funktioniert nach einem einfachen Frage-Antwort-Prinzip

Die aktive (fragende) Klasse stellt in ihrem öffentlichen Header ein **Protokoll** zur Verfügung, das die verwendeten Methoden definiert. Außerdem besitzt sie ein öffentliches Attri-

but, häufig `delegate` genannt, das eine Referenz zu einem Objekt der passiven (antwortenden) Klasse hält. Diese implementiert die Methoden des Protokolls, die in entsprechenden Situationen von der aktiven Klasse aufgerufen werden können.

Methoden in einem Protokoll dienen in den meisten Fällen einem von zwei Zwecken:

- Von dem Delegate Objekt werden Informationen angefordert, oder
- Das Delegate Objekt wird benachrichtigt, wenn eine bestimmte Situation eintritt.

Viele Klassen in Frameworks implementieren Kommunikationsmechanismen in Form von Delegates. Dazu gehört bspw. die im folgenden Abschnitt beschriebene `UITableView`. Wir können jedoch auch eigene Protokolle und Delegates definieren. Betrachten wir dazu das Beispiel eines View Controllers `SelectionViewController : UIViewController`, der zur Auswahl eines bestimmten Objekts dienen soll.

## Aktive Klasse

```

1 // SelectionViewController.h
2
3 @protocol SelectionProtocol; // Forward Declaration: Das Protokoll wird später
   definiert, muss der Klasse an dieser Stelle jedoch schon zur Verfügung
   stehen
4
5 @interface SelectionViewController : UIViewController
6
7 @property (weak, nonatomic) id <SelectionProtocol> delegate;
8
9 @end
10
11 @protocol SelectionProtocol
12
13 - (NSArray *)objectsForSelectionViewController:(SelectionViewController *)
   selectionVC;
14
15 @optional
16
17 - (void)selectionViewController:(SelectionViewController *)selectionVC
   didSelectObject:(id)object;
18
19 @end

```

Das Delegate Attribut ist mit `weak` gekennzeichnet, da die Kommunikation einseitig und die Referenz nicht notwendig ist: Die aktive Klasse soll das Delegate Objekt nicht im Speicher halten, wenn es an anderer Stelle nicht mehr benötigt wird.

Weiterhin ist das Delegate Attribut als Empfänger des `SelectionProtocol` markiert. Dem Attribut sollen also nur Objekte zugewiesen werden, die das Protokoll implementieren. Andernfalls wird eine Warnung ausgegeben.

Mit `@protocol` beginnt schließlich die Definition des Protokolls. Die folgenden Methoden müssen vom Delegate implementiert werden. Optionale Methoden, deren Implementierung nicht erwartet wird, können einem `@optional` folgen.

Diese Methoden können nun von der aktiven Klassen auf dem Delegate Objekt aufgerufen werden:

```

1 // SelectionViewController.m
2
3 #import "SelectionViewController.h"
4
5 @implementation SelectionViewController
6
7 - (void)viewDidLoad {
8     [super viewDidLoad];
9
10    NSArray *objects = [self.delegate objectsForSelectionViewController:self];
11    // configure view to display objects ...
12}
13
14
15 - (void)didSelectObject:(id)object {
16    // assume this method is called when an object is selected
17
18    if ([self.delegate respondsToSelector:@selector(selectionViewController:
19        didSelectObject:)]) { // test, if delegate implements the optional
20        [self.delegate selectionViewController:self didSelectObject:object];
21    }
22}
23
24 @end

```

### Passive Klasse

```

1 // ViewController.h
2
3 #import "SelectionViewController.h"
4
5 @interface ViewController : UIViewController <SelectionProtocol>
6
7 @end

```

Im öffentlichen (oder privaten) Interface kann eine Klasse als Empfänger des Protokolls markiert werden. Von dieser Klasse wird nun die Implementierung der Protokollmethoden erwartet.

```

1 // ViewController.m
2
3 #import "ViewController.h"

```

```

4
5 @implementation ViewController
6
7 - (IBAction)showSelectionViewController:(id)sender {
8     // assume this method is called to present the selection VC
9
10    SelectionViewController *selectionVC = [[SelectionViewController alloc]
11        init];
12    selectionVC.delegate = self; // set delegate property
13    [self presentViewController:selectionVC animated:YES completion:nil];
14 }
15
16 - (NSArray *)objectsForSelectionViewController:(SelectionViewController *)selectionVC {
17     // get objects array ...
18
19     return objects;
20 }
21
22
23 - (void)selectionViewController:(SelectionViewController *)selectionVC didSelectObject:(id)object {
24     // do something with the selected object ...
25
26     [selectionVC dismissViewControllerAnimated:YES completion:nil];
27 }
28
29
30 @end

```

Das Delegate Konzept ist häufig hilfreich, um erforderliche Informationen anzufordern. Dabei muss der aktiven Klasse nichts weiter über die passive Klasse bekannt sein als die Implementierung des Protokolls und sie muss diese nicht importieren. Das Konzept ist daher flexibel einsetzbar. Für konzeptionell eng miteinander verbundene Klassen, bei denen diese Asymmetrie nicht notwendig ist, sollten jedoch stattdessen lieber normale Methodenaufrufe verwendet werden.

## 4.7 Table Views & Table View Controller

Aufgrund des eingeschränkten Platzes auf Geräten der iOS Plattform verwenden sehr viele Apps `UIScrollView` oder Subklassen derselben, um Views über die Bildschirmgröße hinaus darzustellen. `UITableView` ist eine solche Subklasse, die in einem Großteil der iOS Apps zu finden ist.

**Table Views** stellen eine Liste von `UITableViewCell`: `UIView` Objekten dar und verwenden den Scrollmechanismus von `UIScrollView`.

Eine Table View ist zunächst in **Sections** unterteilt, die jeweils eine bestimmte Anzahl von **Rows** enthalten. Zur Identifikation einer Zeile werden daher Objekte der `NSIndexPath`

Klasse verwendet, die jeweils ein Attribut `NSInteger section` und `NSInteger row` besitzen.

Es kann zwischen den Darstellungsoptionen `UITableViewStylePlain` und `UITableViewStyleGrouped` für Table Views gewählt werden, die lediglich das Erscheinungsbild der Table View verändern (s. S. 68, Abb. 4.17).

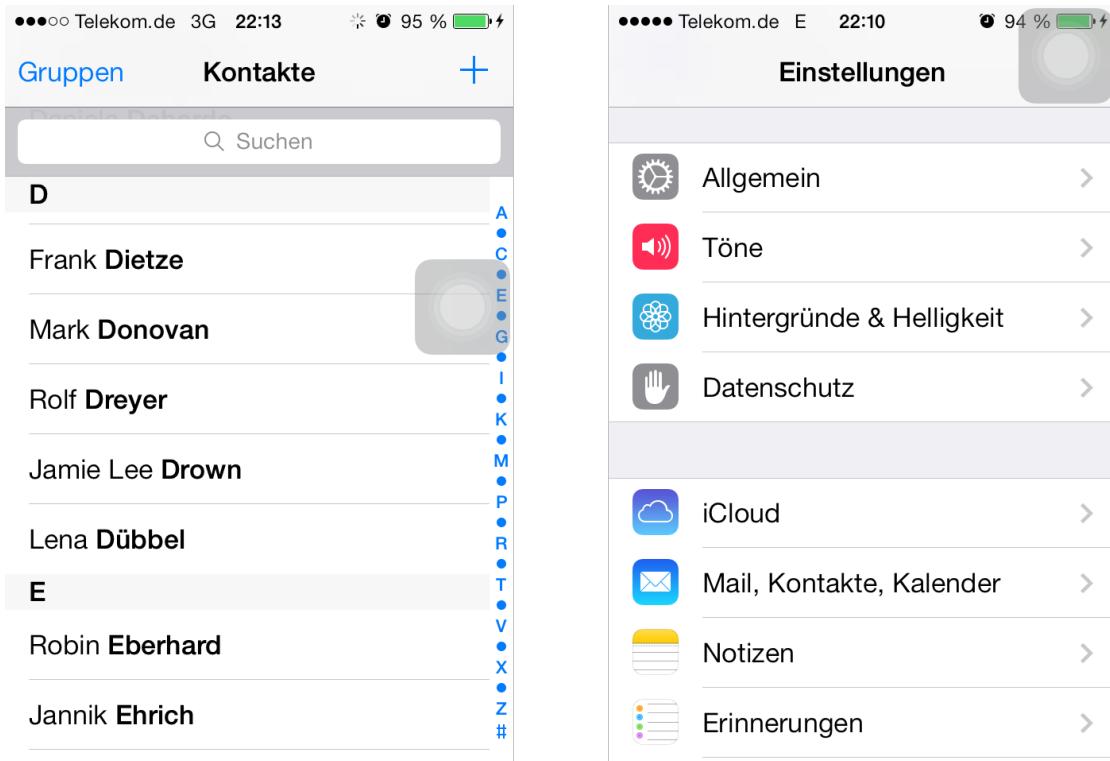


Abbildung 4.17: Table Views können in den Darstellungsoptionen Plain oder Grouped angezeigt werden

#### 4.7.1 Statische und dynamische Table Views

Statische Table Views bieten eine einfache Möglichkeit, Views tabellarisch darzustellen. Im Storyboard kann ein `UITableView` Objekt der View Hierarchie hinzugefügt und mit der Option *Content: Static Cells* versehen werden. Im Attributes Inspector kann die Anzahl der Sections und Rows, sowie eine Vielzahl weiterer Attribute konfiguriert werden. Die `UITableViewCell` Objekte, die jede Zeile repräsentieren, fungieren nun als normale Views, denen wir Subviews in Form von Buttons, Labels o.ä. hinzufügen können.

Häufig möchten wir Table Views jedoch zur Darstellung dynamischer Inhalte verwenden. Dazu stellt `UITableView` nach dem Delegate Konzept die beiden Protokolle `UITableViewDatasource` und `UITableViewDelegate` zur Verfügung und besitzt die entsprechenden Attribute `id <UITableViewDatasource>datasource` und `id <UITableViewDelegate>delegate`. Zur Lauf-

zeit stellt eine dynamische Table View Anfragen an beide Objekte und benachrichtigt diese über Ereignisse.

Anstatt jede `UITableViewCell` im Storyboard einzeln zu konfigurieren verwenden dynamische Table Views **Prototype Cells**. Diese werden im Storyboard wie Static Cells konfiguriert, jedoch erst zur Laufzeit mit Inhalt gefüllt (s. S. 69, Abb. 4.18). Natürlich eignet sich auch an dieser Stelle Auto Layout zur Positionierung der Views. Außerdem sollte jeder Prototype Cell ein Identifier zugewiesen werden.

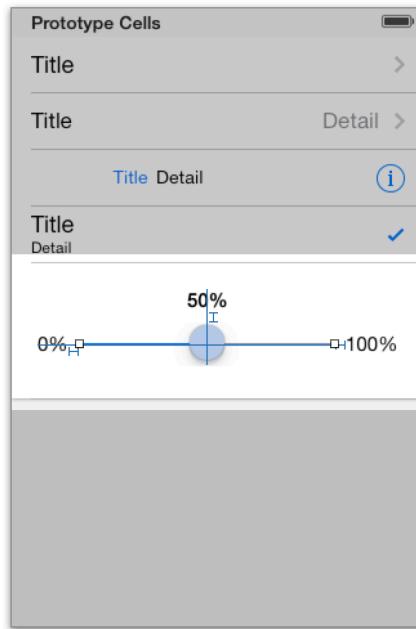


Abbildung 4.18: Prototype Cells können im Storyboard konfiguriert werden und dienen anschließend als Vorlage für dynamische Table View Cells

An der entsprechenden Stelle im Code können wir dann mit der Instanzmethode `dequeueReusableCellWithIdentifier:` von `UITableView` ein `UITableViewCell` Objekt nach Vorlage der Prototype Cell mit dem angegebenen Identifier anfordern und anschließend mit Inhalt füllen. Dabei verwendet `UITableView` einen sehr effektiven Verwaltungsmechanismus, der zunächst solche `UITableViewCell` Objekte wiederverwendet, die gerade nicht angezeigt werden. So kann ein reibungsloses Scrollen durch beliebig lange Listen realisiert werden.

#### 4.7.2 Datasource, Delegate und Table View Controller

Zur Darstellung dynamischer Inhalte verwenden wir meist einen View Controller sowohl als Datasource als auch Delegate einer Table View und implementieren die Methoden der beiden zugehörigen Protokolle.

Prinzipiell kann die Table View an einer beliebigen Stelle der View Hierarchy der Content View eines View Controllers positioniert sein. Wurde ihr der View Controller als Datasource und Delegate zu gewiesen, bspw. mithilfe von IBOutlets im Storyboard, muss dieser die `UITableViewDatasource` und `UITableViewDelegate` Protokolle implementieren.

```
1 @interface ViewController : UIViewController <UITableViewDatasource,
    UITableViewDelegate>
```

In den meisten Fällen verwenden wir jedoch die Table View selbst als Content View des View Controllers. UIKit stellt hierzu die vereinfachende Subklasse `UITableViewController` : `UIViewController` zur Verfügung, die das Content View Attribut `UIView*view` mit `UITableView*tableView` ersetzt und darüber hinaus einige hilfreiche Mechanismen implementiert.

In der Implementierung der `UITableViewController` Subklasse können wir nun die Methoden der Datasource und Delegate Protokolle implementieren. Die Dokumentation beschreibt die Protokolle in gewohnter ausführlicher Form.

Folgende Methoden sollten jedoch in keiner `UITableViewDatasource` Implementierung fehlen:

```
1 // Sei NSArray *objects ein Attribut dieser Subklasse
2
3 - (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
4     // Bestimmt die Anzahl der Sections
5     return 1;
6 }
7
8 - (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
9     // Bestimmt die Anzahl der Rows für die angegebene Section
10    return [self.objects count];
11 }
12
13 - (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
14
15     // Greift auf das Model zurück, um das darzustellende Objekt zu erhalten
16     NS0bject *object = [self.objects objectAtIndex:indexPath.row];
17
18     // Erstellt ein neues UITableViewCell Objekt entsprechend der im Storyboard
19         definierten Prototype Cell mit dem angegebenen Identifier oder
20         verwendet eine existierende, momentan nicht verwendete Zelle mit
21         diesem Identifier
22     UITableViewCell *cell = [tableView dequeueReusableCellWithIdentifier:@"
23         objectCell"];
24
25     // Konfiguriert die View entsprechend des Models
26     cell.textLabel.text = object.description;
27
28     return cell;
```

25 }

An dieser Stelle wird das Model-View-Controller Konzept (*s. S. 50, Abschnitt 4.2*) offensichtlich in Perfektion umgesetzt. Mechanismen dieser Art sollten bei der Konzeption von iOS Apps immer als Referenz für den eigenen Code dienen.

Das **UITableViewDelegate** Protokolls bietet zusätzlich zu vielen weiteren Darstellungsoperationen die Möglichkeit zur Reaktion auf verschiedene Ereignisse. Dazu gehört die vierte essentielle Methode der Table View Programmierung:

```

1 - (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
2
3     // do something ...
4
5 }
```

Alternativ zur Implementierung dieser Delegate Methode können auch Storyboard Segues in Verbindung mit Prototype Cells verwendet werden. In diesem Fall ist es sinnvoll, zunächst den Index Path der betätigten Table View Cell in der **prepareForSegue:sender:** Methode auszulesen und damit bspw. Zugriff auf das entsprechende Model Objekt zu erhalten:

```

1 - (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
2
3     NSIndexPath *indexPath = [self.tableView indexPathForCell:sender];
4
5     // get model object and configure destination view controller ...
6
7 }
```

## 4.8 Data Persistence

Aufgrund der Natur des Multitaskings auf der iOS Plattform kann der Ausführungsstatus einer App (*s. S. 46, Abschnitt 4.1.1*) häufig und für den Entwickler unvorhergesehen wechseln. Der Benutzer der App erwartet dabei, dass die Benutzeroberfläche bei jedem Start der App wiederhergestellt wird und Eingaben erhalten bleiben.

Außerdem sind Benutzereingaben essentieller Bestandteil vieler Apps, die die eingegebenen Daten präsentieren und weiterverarbeiten. Die Speicherung von Daten über die Ausführung der App hinaus ist somit fast immer notwendig.

Durch einige Anpassungen ist es auf der iOS Plattform außerdem möglich, in dieser Form gespeicherte Daten über Apple's iCloud Service auf allen iOS Geräten des Benutzers zur Verfügung zu stellen.

Je nach Konzeption der App wird in den meisten Fällen auf folgende Mechanismen zur Datenspeicherung zurückgegriffen.

### 4.8.1 User Defaults

Häufig benötigen Apps lediglich die Speicherung einiger Parameter, Einstellungen und Benutzereingaben. Einfache Benutzerdaten können mithilfe der `NSUserDefaults` Klasse gespeichert und über die Ausführung der App hinaus zu einem späteren Zeitpunkt wieder aufgerufen werden.

`NSUserDefaults` implementiert die Klassenmethode `standardUserDefaults`, die immer das gleiche Objekt zurückgibt (sog. Singleton Architektur). Die Klasse ähnelt einem `NSDictionary` und implementiert analog nach dem Key-Value Prinzip die Instanzmethoden `setObject:ForKey:` und `objectForKey:`:

```
1 [ [NSUserDefaults standardUserDefaults] setObject:@"Alice" forKey:@"user_name"
    ]; // speichern
2 NSString *userName = [ [NSUserDefaults standardUserDefaults] objectForKey:@"user_name"]; // auslesen
```

Es können nur Objekte der Klassen `NSString`, `NSNumber`, `NSDate` und `NSData` oder ausschließlich mit solchen Objekten gefüllte Instanzen von `NSArray` und `NSDictionary` in dieser Form gespeichert werden.

Eigene Subklassen müssen daher zunächst zu `NSData` Objekten serialisiert werden. Wird dies notwendig, sollte jedoch bereits über die Integration von Core Data nachgedacht werden.

Für einige Klassen wie `UIImage` stehen Methoden zur Serialisierung zur Verfügung:

```
1 NSData *imageData = UIImagePNGRepresentation(image); // Sei UIImage *image
    hier ein Bildobjekt
2 [ [NSUserDefaults standardUserDefaults] setObject:imageData forKey:@"image_data"
    ]; // speichern
3 NSData *imageData = [ [NSUserDefaults standardUserDefaults] objectForKey:@"image_data"];
    // auslesen
4 UIImage *image = [UIImage imageWithData:imageData];
```

Schließlich ist erwähnenswert, dass für die in User Defaults verwendeten Keys (und ebenso für andere ähnliche Mechanismen) häufig **Konstanten** verwendet werden. So werden Tippfehler vermieden, einfache Umbenennungen möglich und die Konsistenz des Codes erhöht. Eine Konstante kann mit dem Befehl `#define` definiert werden. Häufig geschieht diese Definition im Prefix Header (.pch Datei), die von jeder Klasse des Projekts automatisch eingebunden wird.

```
1 // im Prefix Header
2 #define kUserDefaultsKeyUsername @"user_name"
3
4 // an beliebiger Stelle im Code
5 [ [NSUserDefaults standardUserDefaults] setObject:@"Alice" forKey:
    kUserDefaultsKeyUsername]; // speichern
6 NSString *userName = [ [NSUserDefaults standardUserDefaults] objectForKey:
    kUserDefaultsKeyUsername]; // auslesen
```

### 4.8.2 Core Data

Gehen die Anforderungen der App über den User Defaults Mechanismus hinaus, wird das Core Data Framework verwendet. Dieses stellt eine vollständige, auf SQLite basierende Datenbankimplementierung in Objective-C dar.

Xcode stellt hier in Analogie zum Interface Builder einen Editor zur Verfügung, mit dem zunächst eine Datenstruktur konfiguriert werden kann (s. S. 73, Abb. 4.19). Anschließend werden Subklassen von `NSManagedObject`: `NSObject` verwendet, die Elemente der Datenstruktur als Objective-C repräsentieren und von der Datenbank verwaltet werden.

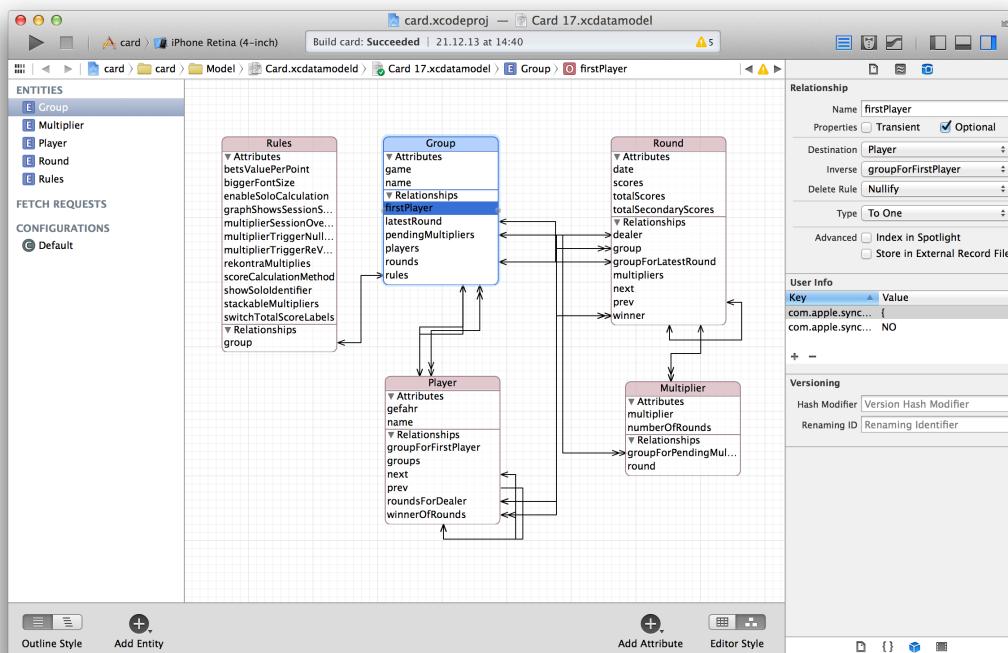


Abbildung 4.19: Core Data Models können in Analogie zum Interface Builder in Xcode graphisch bearbeitet werden

Core Data stellt die Grundlage vieler iOS Apps dar und ist eine sehr komplexe und mächtige Technologie, auf die wir an dieser Stelle nicht im Detail eingehen können. Es ist sehr zu empfehlen, sich bei Bedarf mit Core Data auseinanderzusetzen, anstatt zu versuchen, einen eigenen Mechanismus zur Datenspeicherung zu implementieren. Online sind einige hervorragende Einführungen in Core Data zu finden, von denen ich besonders die Core Data Lecture von Dr. Brad Larson auf iTunes U [4] empfehlen kann, auch wenn sich seit 2010 bereits viele Details verändert haben. Änderungen und Best Practices werden jährlich von

<sup>4</sup><https://itunes.apple.com/de/podcast/7.-core-data/id407243028?i=89378853&mt=2>

Apple's Entwicklern auf der WWDC vorgestellt (*s. S. 7, Abschnitt 1.5*). Ich stehe bei Fragen natürlich ebenfalls gerne zur Verfügung.

### 4.8.3 State Preservation

UIKit bietet ein sehr einfach zu integrierendes System zur Wiederherstellung der Benutzeroberfläche, das jedoch konzeptionell nicht zur Speicherung von Daten den Model Komponente dienen soll. Das State Preservation System ist damit der View- und View Controller Komponente zugeordnet, während für die Model Komponente häufig Core Data verwendet wird.

Elemente der Benutzeroberfläche werden hier anhand des `NSString*restorationIdentifier` Attributs identifiziert. Bei der Wiederherstellung wird für jeden dieser Identifier ein entsprechendes Objekt angefordert, oder bei Verwendung eines Storyboards automatisch erstellt. View Controller und Views können dann die `encodeRestorableStateWithCoder:` und `decodeRestorableStateWithCoder:` Methoden implementieren, um ihre Darstellung zu archivieren und wiederherzustellen. Im iOS App Programming Guide [5] ist das State Preservation System ausführlich dokumentiert.

## 4.9 Notifications

Zur Kommunikation zwischen Klassen haben wir uns bereits mit dem Delegate Konzept beschäftigt (*s. S. 64, Abschnitt 4.6*). Dieses geht von einer direkten Beziehung zwischen einem aktiven (fragenden) Objekt und dessen passiven (antwortenden) Delegate Objekts aus, um Informationen auszutauschen.

Mit **Notifications** steht im Foundation Framework ein weiteres Kommunikationskonzept zur Verfügung, das auf der Verteilung von ungerichteten Benachrichtigungen basiert. Ein (aktives) Objekt versendet hier eine Nachricht mit einem eindeutigen Key, ohne die Empfänger der Nachricht anzugeben. Stattdessen können sich andere (passive) Objekte als Empfänger bestimmter Nachrichten registrieren, um auf diese reagieren zu können.

Dazu implementiert das Foundation Framework die Klasse `NSNotificationCenter` mit der Klassenmethode `defaultCenter` nach dem Singleton Prinzip. Objekte können die Instanzmethode `addObserver:selector:name:object:` nutzen, um bestimmte Nachrichten zu empfangen, oder mit der Instanzmethode `postNotification:` eigene Nachrichten senden.

Beispielsweise wird der Notifications Mechanismus häufig verwendet, um auf Änderungen der User Defaults (*s. S. 72, Abschnitt 4.8.1*) zu reagieren:

---

<sup>5</sup><https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphonеosprogrammingguide/>

#### 4 iOS App Architektur

```
1 // an beliebiger Stelle im Code, bspw. in der viewDidLoad Methode eines View  
2 Controllers  
3 [[NSNotificationCenter defaultCenter] addObserver:self selector:@selector(  
4 userDefaultsDidChange:) name:NSUserDefaultsDidChangeNotification object  
5 :[NSUserDefaults standardUserDefaults]];  
6  
7 // in der Implementierung des View Controllers  
8 - (void)userDefaultsDidChange:(NSNotification *)notification {  
9     // update user interface ...  
10 }
```

# 5 Entwicklungsprozess von iOS Apps

## 5.1 Versionskontrolle mit Git

Die Entwicklung von iOS Apps ist wie jedes Programmierprojekt ein fortschreitender Prozess. Während an Features und Mechaniken im Code gearbeitet wird, besteht immer wieder die Notwendigkeit, vorhandenen Code zu editieren und umzustrukturen. Dabei bietet **Versionskontrolle** (SCM / Software Configuration Management) u.a. die Möglichkeit, diese Änderungen in regelmäßigen Abständen zu sichern, zu vorherigen Versionen zurückzukehren und sogar an verschiedenen Versionen gleichzeitig und mit anderen zusammen zu arbeiten.

Ein sehr beliebtes System der Versionskontrolle ist **Git**. Git wird automatisch mit Xcode installiert und ist ein Kommandozeilenprogramm, auf das wir auf dem Mac mit der Terminal App zugreifen können.

### 5.1.1 Grundlagen der Kommandozeilsyntax

**Navigation** `cd path/to/folder` navigiert zu dem angegebenen Pfad. Die Tilde ~ repräsentiert hier den Benutzerordner.

**Ordnerinhalt** `ls` listet den Inhalt des aktuellen Ordners auf. Mit `ls -a` werden auch versteckte Dateien angezeigt.

**Dateien** `touch filename` erstellt eine neue Datei mit dem angegebenen Dateinamen, während `mkdir dirname` einen Ordner erstellt. `rm filename` löscht die angegebene Datei und `rm -r dirname` den angegebenen Ordner.

### 5.1.2 Git Repository & Commits

Während wir an unserem Projekt arbeiten, können wir Git verwenden, um unseren Code möglichst häufig in Form von **Commits** zu sichern. Dabei werden alle Änderungen an den Projektdateien, die seit dem letzten Commit durchgeführt wurden, in einem versteckten `.git/` Verzeichnis, dem **Repository**, hinterlegt.

Bevor wir Commits sichern können, muss das Repository angelegt werden. Dazu navigieren wir im Terminal in den Projektordner und führend die Initialisierung durch:

```

1 cd path/to/project
2 git init
3 >> Initialized empty Git repository in path/to/project/.git/
4 git status

```

`git status` ist sehr nützlich, um häufig die Situation des Repositories zu prüfen.

Nun können wir Dateien in unserem Projekt verändern, indem wir Code schreiben oder löschen. Mit `git status` sehen wir jederzeit, welche Dateien sich in Bezug auf den vorherigen Commit verändert haben. Befindet sich der Code in einem akzeptablen Zwischenzustand, können wir die Änderungen mit einem Commit im Repository sichern:

```

1 git add --all
2 git commit -m "Commit Message"
3 >> [master (root-commit) a74833f] Commit Message
4 >> x files changed, y insertions(+), z deletions(-)
5 git log

```

Hier ist zu beachten, dass die zu sichernden Änderungen dem anstehenden Commit zunächst mit `git add filename` einzeln oder mit `git add --all` zusammen hinzugefügt werden müssen. `git commit` führt den Commit anschließend durch und erwartet einen String als kurze Beschreibung der Änderungen des Commits. `git log` kann verwendet werden, um eine Liste der letzten Commits im Terminal auszugeben.

Noch nicht in einem Commit gesicherte Änderungen können mit `git reset` verworfen werden:

```
1 git reset --hard
```

### 5.1.3 Branches

Git bietet weiterhin die sehr nützliche Möglichkeit, verschiedene Versionen eines Projekts zu verwalten, indem sich die Commitfolge an einer beliebigen Stelle verzweigt. Dazu können wir mit `git branch` einen neuen **Branch** erstellen. Es kann jederzeit mit `git checkout` zwischen Branches gewechselt werden. Dabei passt Git die Dateien und deren Inhalt im Repository automatisch an.

```

1 git branch new_feature
2 git checkout new_feature
3 # Abkürzung:
4 git checkout -b new_feature

```

Erstellen wir nun weitere Commits, werden diese dem aktiven Branch hinzugefügt.

Um Branches wieder zu vereinigen, bietet Git die **Merge** und **Rebase** Mechanismen. Dabei bestimmt `git merge` die Unterschiede der beiden Branches und erstellt einen Commit, der diese dem aktuellen Branch zusammenfassend hinzufügt. `git rebase` verändert dagegen

die Commitfolge des aktuellen Branches dahingehend, dass die Commits beider Branches so kombiniert werden, als wären sie nacheinander entstanden.

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

Git versucht bei einem Merge oder Rebase, die Änderungen der Branches zu vereinigen. Treten dabei Konflikte auf, wird der Vorgang unterbrochen und die Konflikte müssen zunächst im Code behoben werden. An den entsprechenden Stellen im Code sind dann Markierungen zu finden, die über eine projektübergreifende Suche in Xcode schnell gefunden werden.

```
1 <<<<< HEAD:
2 // alter Code
3 =====
4 // veränderter Code
5 >>>>>
```

Sobald die Konflikte behoben wurden, kann der Vorgang wieder aufgenommen werden:

```
1 git add --all
2 git commit
```

### Feature Branches

In dieser Form werden bspw. sehr häufig **Feature Branches** verwaltet. Dabei wird die stabile und häufig veröffentlichte Version des Projekts von dem `master` Branch des Repositories repräsentiert. Für neue Features oder Umstrukturierungen wird dann eine Branchstruktur angelegt. So kann gearbeitet werden, ohne dass die stabile Version des Projekts verändert wird. Erreicht ein Branch einen stabilen Status und soll veröffentlicht werden, wird ein Merge mit dem `master` Branch durchgeführt. Der Arbeitsbranch kann dabei jederzeit gewechselt werden. So kann bspw. schnell ein Fehler in der veröffentlichten Version in `master` behoben und anschließend wieder zum Feature Branch gewechselt werden.

#### 5.1.4 Zusammenarbeit mit Git & GitHub

Git Repositories ermöglichen die reibungslose Zusammenarbeit mehrerer Entwickler an einem Projekt und ermöglichen dadurch erst die Entwicklung vieler komplexer Projekte, an denen Programmierer auf der ganzen Welt zusammenarbeiten.

Befindet sich eine Kopie des Repositories auf einem Server, kann einem Branch ein serverseitiges Gegenstück zugewiesen werden. Mit `git push` und `git pull` können dieses **Remote Repository** und die lokale Kopie abgeglichen werden.

`git pull` besteht dabei prinzipiell zunächst aus einem Aufruf von `git fetch`, der die serverseitigen Änderungen herunterlädt, und einem anschließenden `git merge`, um die Änderungen in den lokalen Branch zu integrieren.

Die Git Dokumentation (*s. S. 81, Abschnitt 5.1.7*) enthält detaillierte Beschreibungen zu Remote Repositories.

An dieser Stelle darf der Service **GitHub**<sup>[1]</sup> nicht unerwähnt bleiben, der Entwicklern eine Plattform für ihre Git Repositories bietet und für öffentliche Projekte kostenlos ist.

Ein auf GitHub erstelltes Repository kann mit `git clone` heruntergeladen werden, wobei den local Branches automatisch ihr entsprechendes remote Gegenstück zugewiesen wird. Anschließend können wir mit dem local Repository arbeiten und Commits mit dem remote Repository abgleichen.

Entwickler können mit ihrem GitHub Account gemeinsame Repositories erstellen oder solche anderer Entwickler weiterentwickeln. Letztere Mechanik wird **Fork** genannt und bietet die Möglichkeit, an einem Repository eines anderen Entwicklers zu arbeiten und diesem anzubieten, die erstellten Commits in sein Repository zu integrieren. Dazu kann eine **Pull Request** versandt werden, die der Besitzer des Repositories zunächst annehmen muss, damit die Änderungen in sein Repository übernommen werden.

### 5.1.5 Git in Xcode

Versionskontrolle ist tief in Xcode integriert und in vielen Kontextmenüs und Schaltflächen präsent. Das Menü `Source Control` stellt bspw. einige Benutzeroberflächen zu Git Befehlen zur Verfügung, die die Kommandozeilenbedienung ersetzen können. Sehr hilfreich ist jedoch hauptsächlich der Version Editor (*s. S. 80, Abb. 5.1*), der mit der Schaltfläche rechts in der Toolbar alternativ zum Assistant Editor angezeigt werden kann. Im rechten Editorbereich wird dann die Version der links angezeigten Datei zu einem früheren Commit angezeigt und Änderungen visualisiert. Mit der Uhr-Schaltfläche unten in der Mittelleiste können wir frühere Commits auswählen.

### 5.1.6 Gitignore

Elemente können von der Erfassung durch Git ausgenommen, wenn es sich bspw. um benutzerspezifische oder temporäre Dateien handelt. Dazu wird dem Repository eine `.gitignore` Datei hinzugefügt.

- <sup>1</sup> `touch .gitignore`
- `open .gitignore`

---

<sup>1</sup><http://www.github.com>

## 5 Entwicklungsprozess von iOS Apps

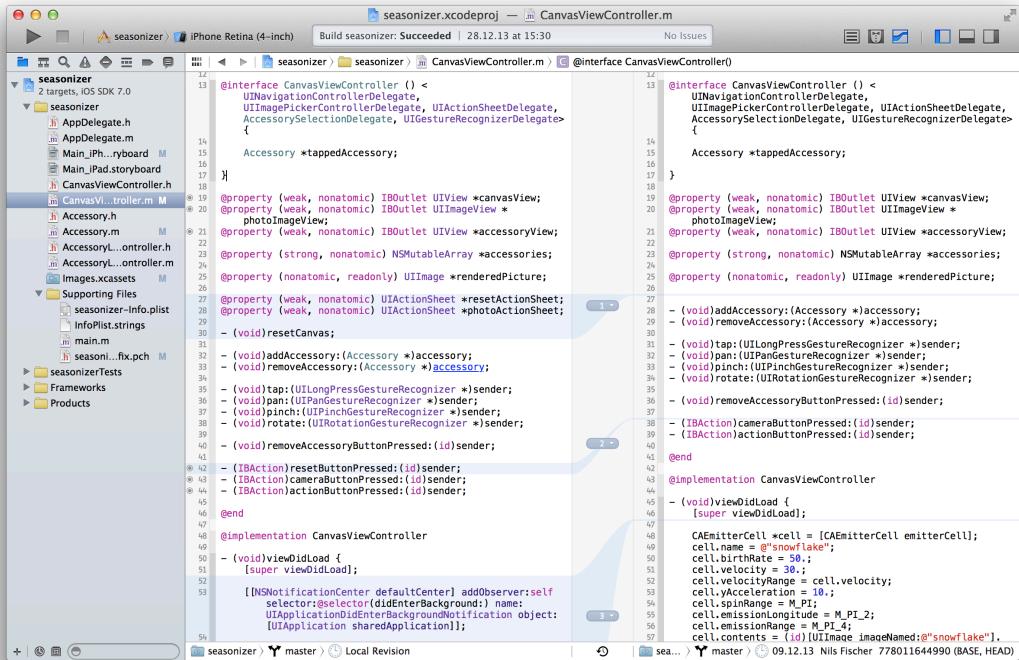


Abbildung 5.1: Der Version Editor zeigt die links geöffnete Datei zu einem früheren Commit an und visualisiert Änderungen

Mit einem Texteditor wie derTextEdit App oder `vim` in der Konsole können wir diese Datei nun konfigurieren. Für Xcode Projekte bietet sich folgende Vorlage an:

```

1 # Mac
2 .DS_Store
3
4 # Xcode
5 */build/*
6 *.pbxuser
7 !default.pbxuser
8 *.mode1v3
9 !default.mode1v3
10 *.mode2v3
11 !default.mode2v3
12 *.perspectivev3
13 !default.perspectivev3
14 xcuserdata
15 profile
16 *.moved-aside
17 DerivedData
18 .idea/
19 *.hmap

```

```
20 *.xcccheckout  
21  
22 #CocoaPods  
23 Pods
```

Die `.gitignore` Datei muss dem Repository zunächst hinzugefügt werden, bevor sie wirksam wird.

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

### 5.1.7 Dokumentation

Git [<sup>2</sup>] ist erstklassig dokumentiert und bietet neben Tutorials und einem Übungsmodus [<sup>3</sup>] eine umfassende Dokumentation [<sup>4</sup>]. Hier sollte bei Bedarf unbedingt nachgeschlagen werden.

---

<sup>2</sup><http://git-scm.com>

<sup>3</sup><http://try.github.com/>

<sup>4</sup><http://git-scm.com/book>

## 5.2 Design & Konzeption von iOS Apps

Eleganter und sinnvoll in Form der bereits diskutierten Architekturkonzepte strukturierter Programmcode bedingt natürlich die Performance unserer App, letztendlich sieht der Benutzer jedoch allein deren User Interface und beurteilt die Funktionsweise anstatt der Architektur.

Aufgrund der mobilen Natur der iOS Plattform und der einfachen Handhabung von Appinstallationen steht (gerade kostenlosen) Apps nur eine begrenzte Zeit zur Verfügung, den Benutzer zu überzeugen. Selten nehmen sich Benutzer die Zeit, sich mit den Funktionen einer App im Detail auseinanderzusetzen, wenn die Benutzerführung nicht sofort intuitiv klar ist.

Dabei wird sich bspw. kaum ein Benutzer über eine leicht verlängerte Laufzeit eines Prozesses aufgrund eines ineffektiven Algorithmus beschweren, solange ihm der Fortschritt auf dem Bildschirm präsentiert wird. Eine 'eingefrorene' App, die jedoch auf effektivste Weise arbeitet, stört das Benutzererlebnis dagegen enorm und wird schnell wieder gelöscht.

Auch auf der iOS Plattform ist Sturgeon's Gesetz deutlich erkennbar: "*ninety percent of everything is crap*". Erfolgreiche Apps sind daher sorgfältig konzipiert und getestet und zeichnen sich durch eine intuitive Benutzerführung aus. Ein Benutzer sollte jederzeit ohne Nachzudenken wissen, wie er handeln muss, um eine bestimmte Wirkung hervorzurufen. Dieses generelle Designkonzept ist nicht auf iOS Apps beschränkt sondern kann auch in vielen anderen Kontexten beobachtet werden (s. S. ??, Abb. ??).

### 5.2.1 App Statement

Beim Konzeptionsprozesses einer App kann es helfen, die wesentlichen Ideen bereits zu Beginn in einem Satz zu charakterisieren. Während der Entwicklung dient dieses **App Statement** anschließend stets der Orientierung und hilft, den Fokus auf die wesentlichen Elemente nicht zu verlieren.

Für die meisten erfolgreichen Apps lässt sich ein solches App Statement ohne Probleme erfassen, da sie auf die Handhabung einer bestimmten Situation ausgerichtet sind. Anhand einiger auf euren Geräten installierter Apps könnt ihr dies schnell überprüfen.

Betrachten wir bspw. die Shazam App<sup>5</sup>, könnte das App Statement lauten:

*Shazam erkennt den in der Umgebung laufenden Song und hilft damit bei der Entdeckung neuer Titel.*

Die Benutzeroberfläche der App ist entsprechend dieser Aussage auf das Wesentliche beschränkt und Bedarf keiner weiteren Erklärung (s. S. 83, Abb. 5.2). Wurde ein Song erkannt, werden weitere Optionen eingeblendet, doch die Erkennfunktion ist weiterhin jederzeit

---

<sup>5</sup><https://itunes.apple.com/de/app/shazam/id284993459?mt=8>

## 5 Entwicklungsprozess von iOS Apps

mit einem Tipp zu Erreichen, während der Benutzer bei Bedarf weitere Bereiche der App ausprobieren kann.



Abbildung 5.2: Die Benutzeroberfläche der Shazam App Bedarf keiner weiteren Erklärung

### 5.2.2 Mockups

Nach der Verfassung eines App Statements gilt es, Benutzeroberfläche und Datenstrukturen zu konzeptionieren. Für beide eignet sich tatsächlich der traditionelle **Bleistift** am besten, um auf Papier Skizzen anzufertigen.

Anschließend lassen sich anhand dieser dann sehr einfach Storyboards und Core Data Models erstellen.

### 5.2.3 Human Interface Guidelines

Jeder iOS Entwickler sollte die **Human Interface Guidelines** [6] gelesen haben und bei Designentscheidungen zu Rate ziehen. Sie geben Aufschluss über die Designkonzepte in iOS und helfen, UIKit Elemente im richtigen Kontext zu verwenden.

---

<sup>6</sup><https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/>

## 5.3 Veröffentlichung im iOS App Store

Am Ende des initialen Entwicklungsprozesses einer iOS App steht häufig die Veröffentlichung im App Store. Darüber sollte sich bereits während der Entwicklung Gedanken gemacht werden, sodass der Veröffentlichung keine organisatorischen Hürden im Weg stehen.

### 5.3.1 Entwickleraccount

Während ihr im Rahmen dieses Kurses Teil des iOS Developer Teams der Uni Heidelberg seid, benötigt ihr für die Veröffentlichung eurer Apps im App Store einen eigenen Entwickleraccount. Mit eurer Apple ID könnt ihr dazu dem iOS Developer Program als **Individual** oder **Company** für jeweils etwa 80€ jährlich beitreten [<sup>7</sup>]. Diese unterscheiden sich lediglich in der zusätzlichen Möglichkeit des Company Accounts, ein Developer Team mit mehreren Mitgliedern und ihren Rechten zu verwalten. Arbeitet ihr zunächst allein, ist ein Individual Account ausreichend und kann bei Bedarf mit Kontakt zum Apple Developer Support später in einen Company Account umgewandelt werden.

Während ihr auch ohne einen solchen bezahlten Entwickleraccount Apps schreiben und auf dem Simulator testen könnt, ist er notwendig, um Certificates und Provisioning Profiles zur Installation auf iOS Geräten und zur Veröffentlichung im App Store generieren zu können (s. S. 26, Abschnitt 2.5.1).

### 5.3.2 Unternehmensanmeldung

*Hinweis:* Natürlich bin ich weder Anwalt noch Finanzberater und kann hier allein meine eigene Erfahrung wiedergeben. Wertvolle Informationen zu Themen dieser Art kann hingegen die lokale **IHK / Industrie- und Handelskammer** geben.

Es ist unumgänglich, aber sehr einfach, für die Teilnahme am iOS Developer Program ein Unternehmen anzumelden. Dazu ist lediglich ein Besuch beim nächsten **Gewerbeamt**, die Angabe einiger Daten und die Bezahlung einer geringen Gebühr von etwa 30€ erforderlich.

Ein Name für das Unternehmen muss an dieser Stelle im Allgemeinen nicht angegeben werden. Trotzdem sollte sich natürlich eine Geschäftsbezeichnung überlegt werden. Sobald diese öffentlich verwendet wird, gilt sie als die offizielle Bezeichnung des Unternehmens und unterliegt einem rechtlichen Schutz. Es kann hilfreich sein, sich die Geschäftsbezeichnung von einer öffentlichen Instanz, wie bspw. der lokalen IHK, bestätigen zu lassen. Unbedingt sollte vorher darauf geachtet werden, dass der gewählte Name nicht anderweitig ähnlich verwendet wird, um Konflikte zu vermeiden. Außerdem muss der Name frei von geschützten Begriffen sein, insbesondere Bezeichnungen wie *iPhone* oder *iOS*.

---

<sup>7</sup><https://developer.apple.com/programs/ios/>

Das Dokument der Gewerbeanmeldung sowie der Bescheid über die Erteilung der Steuernummer muss bei der Anmeldung zum iOS Developer Program auf Anfrage eingereicht werden.

Natürlich ist anschließend eine jährliche Steuererklärung einzureichen. Im Rahmen der Kleinunternehmerregelung gibt es hier eine untere Grenze für das zu versteuernde Einkommen, über die das Finanzamt oder die IHK informiert.

### 5.3.3 iTunesConnect

Die Schnittstelle zum iOS App Store stellt die Weboberfläche **iTunes Connect** [8] dar. Hier können sämtliche Metadaten eurer Apps im App Store verwaltet werden. Dazu gehören Anzeigename und -icon, Beschreibung und Screenshots sowie Preis- und Verfügbarkeiteinstellungen.

Mit eurer Apple ID eingeloggt, könnt ihr Apps erstellen und deren Metadaten bearbeiten.

#### Keywords

Da die Suche eine wichtige Entdeckungsquelle jeder App im App Store darstellt, sei an dieser Stelle kurz ein Hinweis zur Vergabe der Keywords gegeben, die Teil der App Metadaten sind.

Keywords werden als kommagetrennte Liste einzelner Begriffe gespeichert und führen zu einem höheren Rang eurer App in den Suchergebnissen, wenn die Suchanfrage eines oder mehrere Keywords enthält. Dabei sind nur einzelne Wörter entscheidend, die nicht in verschiedenen Kombinationen wiederholt werden müssen. Mit den Keywords *photo* und *editor* wäre also das zusätzliche Keyword *photo editor* redundant und kann weggelassen werden.

Außerdem ist zu beachten, dass der App Name entscheidend zum Suchergebnis beiträgt und offenbar höher als Keywords gewertet wird.

#### Binary Upload

Um eine neue App oder ein Update zu veröffentlichen, muss die entsprechende App im iTunes Connect Interface durch

Mit Xcode kann ein **Archive** kompiliert werden, das eine abgeschlossene Version der App darstellt. Wählt dazu *iOS Gerät* als Target und anschließend **Product > Archive**.

Nach dem Kompilieren erscheint das Archive im Organizer, auch aufrufbar mit der Tastenkombination **[⌘]+[⬆]+[2]**. Ein Klick auf *Distribute* öffnet den Dialog zum Upload zu iTunes Connect. Hier kann das entsprechende Distribution Provisioning Profile und der

---

<sup>8</sup><https://itunesconnect.apple.com>

korrespondierende Eintrag in iTunes Connect ausgewählt werden. Letzterer erscheint nur, wenn die App im iTunes Connect Interface zuvor mit einem Klick auf *Ready to upload binary* markiert wurde. Anschließend wird die App Binary auf Konfigurationsfehler geprüft und hochgeladen.

### 5.3.4 App Review

Jede neue App und jedes Update wird zunächst von Apple Mitarbeitern geprüft, bevor es im App Store veröffentlicht wird. Dieser Review Prozess dauert in der Regel nur wenige Tage, kann jedoch häufig zu einer Zurückweisung führen.

In den meisten Fällen handelt es sich bei dem Grund für eine Zurückweisung um einen Fehler in der App, wie bspw. einen Crash oder einen offensichtlichen Anzeigefehler. Solche sollten natürlich einfach behoben und die App mit kurzem Dank für den Hinweis erneut eingereicht werden.

Manchmal führen jedoch auch andere Mängel zu einer Zurückweisung, die nicht sofort offensichtlich aber in den Reviewnotizen beschrieben sind. Eine meiner Apps, die in Zusammenspiel mit einer anderen konzipiert wurde, ist bspw. aufgrund fehlender Funktionalität als eigenständige App zurückgewiesen worden. Nach einer Erklärung meinerseits wurde die App akzeptiert, trotzdem habe ich der App in einem folgenden Update einen klaren Hinweis auf die Funktionalität hinzugefügt und einige Funktionen überdacht. Eine App wird hier von erfahrenen Apple Mitarbeitern getestet und Zurückweisungen sollten als Anlass genommen werden, sich konzeptionell mit den bemängelten Aspekten der App auseinanderzusetzen.

### 5.3.5 Geschäftsmodelle

Zu den Anfangszeiten des iOS App Stores waren häufig Erfolgsgeschichten einzelner Entwickler zu lesen, die mit ihren Apps 'über Nacht' reich wurden. Mittlerweile ist der App Store zu einem stabilen Markt geworden, der neben größeren Unternehmen weiterhin auch unabhängigen Entwicklern die Möglichkeit bietet, ihre Apps ohne großen organisatorischen Aufwand einem internationalen Publikum anzubieten.

Es haben sich verschiedene Geschäftsmodelle etabliert, die jeweils situationsbedingt einzusetzen sind. An dieser Stelle gebe ich einen Überblick über einige Verkaufsstrategien und eigene Erfahrungen. Es sei weiterhin auf das sehr informative Video *App Store Distribution and Marketing for Apps* [9] der iOS 7 Tech Talk Veranstaltung verwiesen, in dem wertvolle Details zu verschiedenen Geschäftsmodellen und ihren Einsatzmöglichkeiten gegeben werden.

---

<sup>9</sup><https://developer.apple.com/tech-talks/videos/>

## Bezahlt

Das konventionelle Geschäftsmodell im App Store ordnet jeder App einen Preis zu. Der Entwickler erhält dabei 70% der Umsätze.

Im iTunesConnect Interface kann zwischen verschiedenen Preiskategorien gewählt werden, die in den App Stores einem vergleichbaren Wert in der lokalen Währung entsprechen.

Die Wahl eines Preises ist nicht einfach und sollte wohlüberlegt sein. Offensichtlich ist, dass ein niedrigerer Preis nahezu in jedem Fall zu mehr Verkäufen führt und ein höherer Preis zu weniger Verkäufen. Dieser Zusammenhang ist jedoch nicht unbedingt proportional und damit der Umsatz als Produkt von Preis und Verkaufszahl nicht konstant. Das Umsatzmaximum zu finden ist schwierig und es spielen viele Faktoren eine Rolle, von denen die Wahl der Zielgruppe sicherlich einer der Wichtigsten ist. Es ist sehr einfach, den Preis einer App jederzeit zu ändern und es bietet sich an, Umsatzänderungen durch regelmäßige Preisanpassungen zu erfassen.

Weit verbreitet ist der Trugschluss, dass Apps in den untersten Preiskategorien platziert sein müssen, um erfolgreich zu sein. Abhängig von der Zielgruppe der App kann ein sinnvoller Preis gewählt werden, der konstante Umsätze generiert.

## Gratis & Werbefinanziert

Zunächst ist es wichtig anzumerken, dass natürlich nicht jede App profitorientiert verkauft wird und es gibt viele Gründe, Apps kostenlos anzubieten.

Viele Unternehmen wie Yelp<sup>[10]</sup>, Flickr<sup>[11]</sup> und natürlich Facebook<sup>[12]</sup> und Twitter<sup>[13]</sup> bieten kostenlose Apps zu ihrem Onlineservice, um ihren Nutzern von Smartphones und Tablets nativen Zugriff auf die angebotenen Funktionen zu ermöglichen. Ähnliche **Companion Apps** werden auch für lokale Unternehmen wie Shops und Restaurants immer beliebter.

Während bei Companion Apps in der Regel andere Formen der Monetarisierung existieren, gibt es auch völlig kostenfreie Apps, zu denen häufig **Hobby- und Open-Source-Projekte** gehören. Es kann sehr sinnvoll sein, eine solche App zu veröffentlichen. Kostenlose Apps werden um ein Vielfaches häufiger heruntergeladen als bezahlte und können bspw. Benutzer auf andere eigene Apps oder Webseiten aufmerksam machen oder als Showcase für potentielle Kunden dienen, um die eigenen Entwicklerfähigkeiten zu demonstrieren.

---

<sup>10</sup><http://www.yelp.de>

<sup>11</sup><http://www.flickr.com>

<sup>12</sup><http://www.facebook.com>

<sup>13</sup><http://www.twitter.com>

Die Integration von **Werbung** in eine kostenlose App ist ein verbreitetes Geschäftsmodell und kann situationsbedingt zu höheren Umsätzen als das konventionelle Bezahlkonzept führen. Mit Apple's iAd Framework ist es sehr einfach, Werbebanner verschiedener Formate in einer App anzuzeigen. Außerdem sind die iAd Banner qualitativ hochwertig, interaktiv und verursachen keinen störenden App-Wechsel. Die schnellste Möglichkeit der Implementierung besteht aus nicht mehr als zwei Zeilen Code in einer View Controller Klasse:

```

1 @import iAd; // Importiere das iAd Framework
2
3 - (void)viewDidLoad {
4     [super viewDidLoad];
5
6     self.canDisplayBannerAds = YES;
7 }
```

Damit das iAd Framework verwendet werden kann, muss es zunächst noch dem Projekt hinzugefügt werden. In der Targetkonfiguration (s. S. 11, Abschnitt 2.1.3) unter **General** > **Linked Frameworks and Libraries** genügt ein Klick auf den **+**-Button, woraufhin das iAd Framework ausgewählt werden kann.

Der iAd Programming Guide [14] enthält viele weiterführende Information, ist aber momentan leider nicht vollständig aktuell. Stattdessen sind die WWDC Videos empfehlenswert, die sich mit Neuerungen und Best Practices befassen, u.a. bezüglich iAd (s. S. 7, Abschnitt 1.5).

Werden in einer App iAd Werbebanner angezeigt, erhält der Entwickler 70% der Einnahmen, die sich aus **Clicks** und **Impressions** zusammensetzen. Dabei stellt jede Präsentation eines Banners in der App eine Impression dar.

## Freemium

Aufgrund der weit höheren Downloadrate kostenloser im Vergleich zu bezahlten Apps, setzen mittlerweile viele Entwickler auf das **Freemium** Geschäftsmodell. Hier wird zunächst ein kostenloser Service angeboten, der mit bezahlten Features erweiterbar ist ("Premium"). In iOS Apps werden dazu **In-App-Käufe** oder **Abonnements** verwendet. Für beide Systeme bietet Apple Frameworks an.

Einige der umsatzstärksten Apps im App Store basieren auf dem Freemium Konzept und zeigen, wie dieses erfolgreich umgesetzt werden kann. Dieses Geschäftsmodell eignet sich nicht für jede App und muss sinnvoll integriert werden. Im o.g. Video *App Store Distribution and Marketing for Apps* wird das Freemium Konzept im Detail erläutert.

---

<sup>14</sup>[https://developer.apple.com/library/ios/documentation/userexperience/conceptual/iAd\\_Guide/](https://developer.apple.com/library/ios/documentation/userexperience/conceptual/iAd_Guide/)

### 5.3.6 Marketing

#### Werbung

Der App Store bietet verschiedene Möglichkeiten für Benutzer, Apps zu entdecken, zu denen bspw. Top-Listen und redaktionell verwaltete Highlights gehören. Da die Chance, in eine solche Liste aufgenommen zu werden, jedoch sehr gering ist, sollte sich das Marketing einer App auf verlässlichere Methoden konzentrieren.

Viele Benutzer verwenden die Suchfunktion, um eine passende App zu finden, und die Wahl geeigneter Keywords und gut gestalteter iTunesConnect Metadaten ist dementsprechend wichtig. Ein konstanter Strom potentieller Käufer auf die App Store Seite einer App wird jedoch erst durch die sinnvolle Platzierung von Werbung möglich.

Selbst als eigenständiger Entwickler ohne großes Marketingbudget sollte über Werbung nachgedacht werden. Einen einfachen Startpunkt dafür bietet bspw. *Google AdWords* [15].

Werbeanzeigen sollten dabei auf eine Webseite verweisen, die die Funktionen der App im Detail beschreibt und bereits zu einer Kaufentscheidung führt. Dazu ist ein Video eines der wichtigsten Hilfsmittel. Klickt der Besucher auf den Link zum App Store und ist bereits zuvor über Preis und Leistung der App informiert, wird er sie in den meisten Fällen kaufen. Der Anteil der Besucher der Webseite, die die App schließlich kaufen, wird **Conversion Rate** genannt. Da keine Informationen vorliegen, wie häufig die App Store Seite einer App besucht wird, kann es schwierig sein, die Conversion Rate zu messen. Mithilfe verschiedener Services wie *Google Analytics* [16] kann sie jedoch abgeschätzt werden, indem eine Korrelation zwischen Webseitenbesuchen und App Käufen hergestellt wird.

Die Conversion Rate dient anschließend als Grundlage für die Berechnung des Werbebudgets. Der maximale Preis, der pro Klick auf eine Werbeanzeige bezahlt werden kann, entspricht dabei dem effektiven Geldwert eines Besuchers auf der Webseite:

$$\frac{Price}{Click} = ConversionRate * \frac{Profit}{Sale} \quad (5.1)$$

Dabei gilt in der Regel aufgrund der Umsatzverteilung im App Store:

$$\frac{Profit}{Sale} = AppPrice * 0.7 \quad (5.2)$$

Da mit einer guten Schätzung der Conversion Rate jeder Klick auf eine Werbeanzeige somit weniger kostet, als er Gewinn bringt, ist eine Werbekampagne in jedem Fall lohnenswert.

Werbung ist damit eine Maschine, die mehr produziert, als sie verbraucht – ganz im Widerspruch zur Thermodynamik.

---

<sup>15</sup><http://adwords.google.com>

<sup>16</sup><http://www.google.com/analytics/>

## Updates

Im App Store besteht ein beobachtbarer Zusammenhang zwischen Veröffentlichung von Updates und Downloadzahlen. Erhält eine App ein Update, werden Benutzer auf sie aufmerksam, verwenden sie häufiger und erzählen ihren Freunden von der App. Ohne Updates oder Werbung sinken die Downloadzahlen stetig.

Es wird empfohlen, etwa alle 4–6 Wochen ein Update zu veröffentlichen, auch wenn es nur aus einigen Fehlerbehebungen oder Optimierungen besteht.

Weiterhin erwarten insbesondere Benutzer der iOS Plattform eine Anpassung ihrer Apps an neue Technologien, wenn diese verfügbar werden. Statistiken zeigen, dass bspw. iOS 7 innerhalb von nur 4 Tagen auf über 50% aller iOS Geräte installiert wurde [17] und neue Mechaniken wie iCloud damit bereits nach kurzer Zeit sehr vielen Nutzern zur Verfügung stehen. In solchen Situationen werden Apps, die diese neuen Technologien bereits unterstützen, bevorzugt heruntergeladen und können einen Marktvorteil erlangen.

### 5.3.7 Statistik

iTunesConnect bietet die Möglichkeit, die Downloadzahlen eigener Apps einzusehen und entsprechende **Reports** der Kategorien *daily, weekly, monthly, yearly und financial* herunterzuladen. Dies kann manuell geschehen, doch es haben sich viele Services entwickelt, die diese Daten aufbereiten, darstellen und archivieren.

Dazu gehören Mac Apps wie *AppViz* [18] und Onlineservices wie *Appfigures* [19], die jedoch selten kostenlos sind. Der Appfigures Service, den ich bereits seit einigen Jahren verwende, bietet bspw. die Möglichkeit, in konfigurierbaren Intervallen eine Email mit aktuellen Statistiken wie Download-, Update- und Profitzahlen sowie Reviews und Rankings zu versenden, sodass man täglich über den Profit des Vortags informiert werden kann.

Wertvolle Informationen, die über die in iTunesConnect verfügbaren Downloadzahlen hinausgehen, kann eine Integration eines Statistik-Frameworks in die eigene App liefern. Analog zu Websitestatistiken stellt *Google Analytics* [20] eine einfach zu implementierende und umfangreiche Möglichkeit dar, Informationen über die Benutzer einer App zu erhalten. Google Analytics stellt nicht nur Daten über Verwendungsdauer, Standort-, Sprach- und Geräteverteilung, Verhaltensflüsse und benutzerdefinierte Events zur Verfügung, sondern zeigt sogar in Echtzeit an, wo Benutzer die App gerade verwenden und welche Aktionen sie ausführen.

Diese Informationen können verwendet werden, um eine App bspw. an bestimmte Bildschirmauflösungen anzupassen, Lokalisierungen für bestimmte Sprachen hinzuzufügen oder

---

<sup>17</sup>[https://mixpanel.com/trends/#report/ios\\_7](https://mixpanel.com/trends/#report/ios_7)

<sup>18</sup><https://appviz.com>

<sup>19</sup><https://appfigures.com>

<sup>20</sup><http://www.google.com/analytics/>

## *5 Entwicklungsprozess von iOS Apps*

die Benutzeroberfläche zu optimieren. Erfordert die App z.B. einen Login, kann mit einem solchen Service analysiert werden, wie viele Benutzer, die die App herunterladen, einen Account erstellen, oder ob neu implementierte Maßnahmen wie Push Notifications zu einer höheren Aktivität der Nutzer führen.