

Universität Heidelberg

Sommersemester 2016

Softwareentwicklung für iOS

Skript

Nils Fischer

Aktualisiert am 5. Juni 2016

Kursdetails und begleitende Materialien auf der Vorlesungswebseite:

<http://ios-dev-kurs.github.io>

Inhaltsverzeichnis

1	Einführung	4
1.1	Über dieses Skript	4
1.2	Programmieren in Swift	4
1.3	Dokumentationen und Referenzen	5
2	Xcode	7
2.1	Workspaces & Projekte	7
2.1.1	Neue Projekte anlegen	7
2.1.2	Targets und Products	9
2.1.3	Projekt- und Target-Konfiguration	10
2.2	Benutzerinterface	12
2.2.1	Darstellungsmodi	13
2.2.2	Build & Run	13
2.2.3	Navigator	14
2.2.4	Editor	14
2.2.5	Inspektor	18
2.2.6	Debug-Bereich & Konsole	18
2.3	Interface Builder	19
2.3.1	XIBs & Storyboards	20
2.3.2	Inspektor im IB-Modus	21
2.3.3	IBOutlets & IBActions	21
2.4	Dokumentation	23
2.5	Testen auf iOS Geräten	24
2.5.1	Der Provisioning Prozess	25
3	Projektstrukturierung	27
3.1	Versionskontrolle mit Git	27
3.1.1	Grundlagen der Kommandozeilensyntax	28
3.1.2	Git Repository & Commits	28
3.1.3	Branches	29
3.1.4	Zusammenarbeit mit Git & GitHub	30
3.1.5	Git in Xcode	31
3.1.6	Gitignore	32
3.1.7	Dokumentation	33



4	iOS App Architektur	34
4.1	iOS App Lifecycle	34
4.1.1	App States	34
4.1.2	Startprozess einer iOS App	35
4.2	Das Model-View-Controller Konzept	38
4.3	View Hierarchie	39
4.3.1	Frame und CGRect	39
4.3.2	UIView Objekte	41
4.4	Auto Layout	41
4.4.1	Constraints	42
4.4.2	Intrinsic Content Size	42
4.4.3	Auto Layout im Interface Builder	43
4.4.4	Auto Layout im Code	45
4.4.5	Size Classes	46
4.5	View Controller Hierarchie	46
4.5.1	View Controller Lifecycle	49
4.5.2	Präsentation von View Controllern	50
4.5.3	Container View Controller in UIKit	50
4.5.4	View Controller in Storyboards	51
4.5.5	Segues	52
4.6	Das Delegate Konzept	54
4.7	Table Views & Table View Controller	57
4.7.1	Statische und dynamische Table Views	57
4.7.2	Datasource, Delegate und Table View Controller	59
4.8	Data Persistence	60
4.8.1	User Defaults	61
4.8.2	Core Data	62
4.8.3	State Preservation	63
4.9	Notifications	63
5	Entwicklungsprozess von iOS Apps	65
5.1	Unit Tests & Test Driven Development	65
5.2	Open Source Libraries & CocoaPods	66

Kapitel 1

Einführung

1.1 Über dieses Skript

Dieses Skript wird im Verlauf des Semesters auf der Vorlesungswebseite ^[1] 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 auf der Vorlesungsseite 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 ( + ) 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.

Die Inhalte basieren auf der Xcode Version 7.3 und dem iOS SDK 9.3. 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.

1.2 Programmieren in Swift

Bis Apple auf der WWDC im Juni 2014 die völlig neue Programmiersprache *Swift* vorstellte, wurde Software für die iOS und Mac Plattformen fast ausschließlich in der auf C basierenden Programmiersprache *Objective-C* geschrieben. Seitdem hat Swift bereits enormen Zulauf erhalten und wird Objective-C langfristig ersetzen.

Swift bedient sich vieler Konzepte moderner Programmiersprachen und legt besonderen Wert auf eine aussagekräftige Syntax und typsicheren, unmissverständlichen Code,

¹<http://ios-dev-kurs.github.io/>

der Laufzeitfehler vermeidet. Swift's Grammatik ist sehr klein gehalten und viele grundlegende Typen, Funktionen und Operatoren sind stattdessen in der Sprache selbst in der *Swift standard library* definiert.

Noch immer befindet sich Swift stark in Entwicklung und neue Versionen werden häufig veröffentlicht. Statt in diesem Skript eine Einführung in diese Sprache zu geben, verweise ich auf das Buch *The Swift Programming Language* von Apple, das sowohl online ^[2] als auch im iBooks Store ^[3] immer in aktueller Version vorliegt und uns in diesem Kurs als Referenz dienen wird.

Zu Beginn des Kurses lernen wir die Grundlagen der objektorientierten Programmierung in Swift anhand dieses Buches und werden dann sehr schnell anfangen, unsere ersten iOS Apps zu schreiben.

1.3 Dokumentationen und Referenzen

Zusätzlich zu der in Xcode integrierten (s. S. 23, Abschnitt 2.4) und online verfügbaren Dokumentation der Frameworks der iOS Plattform bietet Apple einige Ressourcen für iOS Developer an:

iOS Dev Center ^[4] 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. 25, Abschnitt 2.5.1) zu finden.

iOS Human Interface Guidelines (HIG) ^[5] 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 ^[6].

iOS App Programming Guide ^[7] stellt eine Übersicht über die Architektur von iOS Apps dar.

WWDC Videos ^[8] 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.

²https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

³<https://itunes.apple.com/de/book/swift-programming-language/id881256329?mt=11>

⁴<https://developer.apple.com/devcenter/ios/>

⁵<https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/>

⁶<https://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/OSXHIGuidelines/>

⁷<https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/>

⁸<https://developer.apple.com/wwdc/videos/>

Hier darf natürlich auch die Community-basierte Q&A-Seite **Stack Overflow** [⁹] nicht unerwähnt bleiben, die bei Codefragen immer sehr hilfreich ist.

⁹<http://stackoverflow.com>

Kapitel 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. Apple bietet online ^[1] eine umfassende Dokumentation über den Funktionsumfang von Xcode an.

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, Benutzerinterfaces 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 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. 8, Abb. 2.1) ein Template. In diesem Dialogfenster finden wir mit `OS X > Application > Cocoa Application` auch das Template, um Mac OS X Apps zu schreiben. Für iOS Apps verwenden wir die Templates unter `iOS > Application`.

¹https://developer.apple.com/library/mac/documentation/ToolsLanguages/Conceptual/Xcode_Overview/

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.

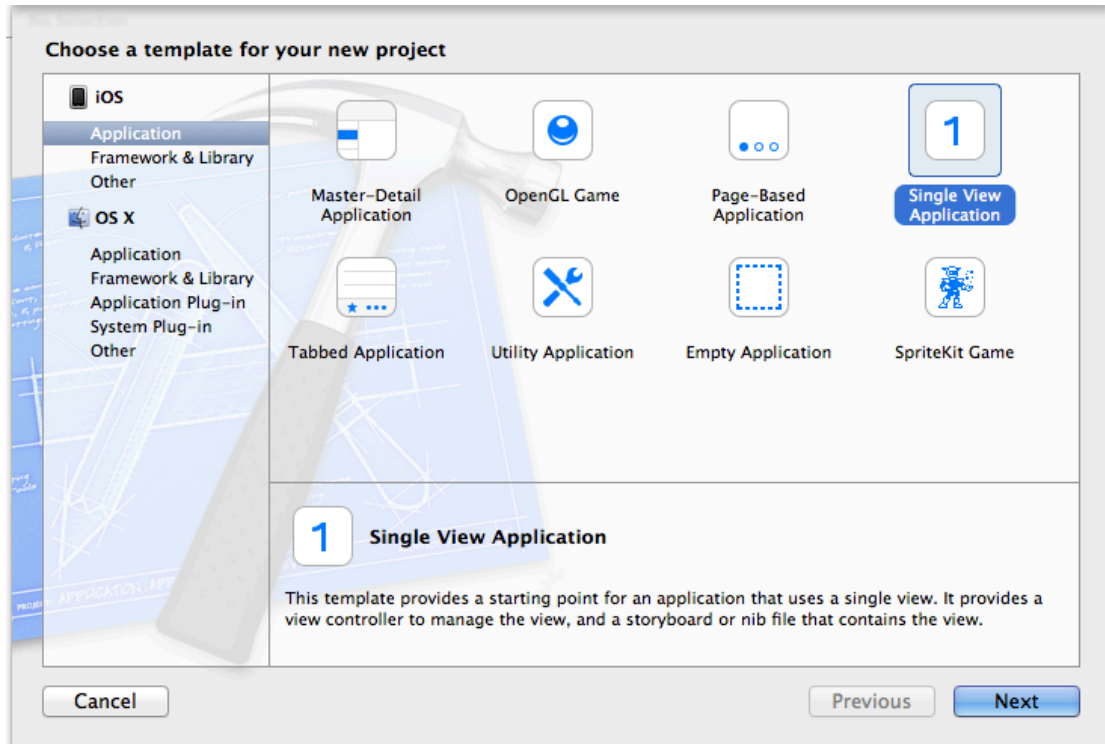


Abbildung 2.1: Ein neues Xcode-Projekt anlegen

Nach der Wahl des Templates werden weitere Optionen für das Projekt präsentiert (s. S. 9, Abb. 2.2).

Product Name identifiziert die resultierende App im Projekt. Es können später in einem Projekt weitere Produkte hinzugefügt werden (s. S. 9, 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 Datei 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.<deiname>**" 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.<deiname>.<productname>**". Der Bundle Identifier identifiziert eine App eindeutig im App Store und auf Apple's Servern.

Language bietet die Auswahl zwischen Swift und Objective-C als Basis-Programmiersprache für das Projekt. Code der anderen Sprache kann ebenfalls eingebunden werden.

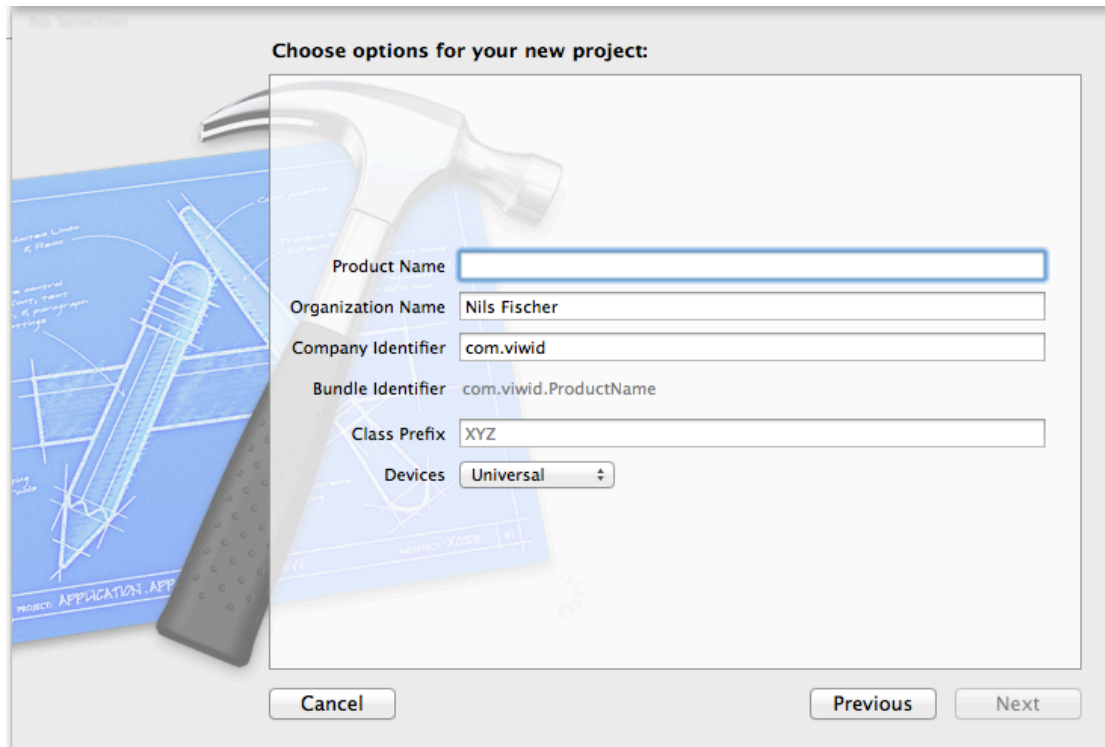


Abbildung 2.2: Ein neues Projekt konfigurieren

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

Use Core Data fügt dem Projekt direkt die benötigten Elemente zur Core Data Integration hinzu. Dies ist eine Datenbanktechnologie auf Basis von SQL (s. S. 62, Abschnitt 4.8.2).

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 ausführlich thematisieren (s. S. 27, Abschnitt 3.1).

2.1.2 Targets und Products

Ein Projekt kann zur Entwicklung mehrerer Apps dienen, die sich Programmcode teilen können.

Das Ergebnis des Compilers, also eine 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, referenzierte Dateien, benötigte Frameworks 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 (s. S. 10, Abb. 2.3). Mit der Schaltfläche oben links kann das Projekt oder ein Target ausgewählt werden.

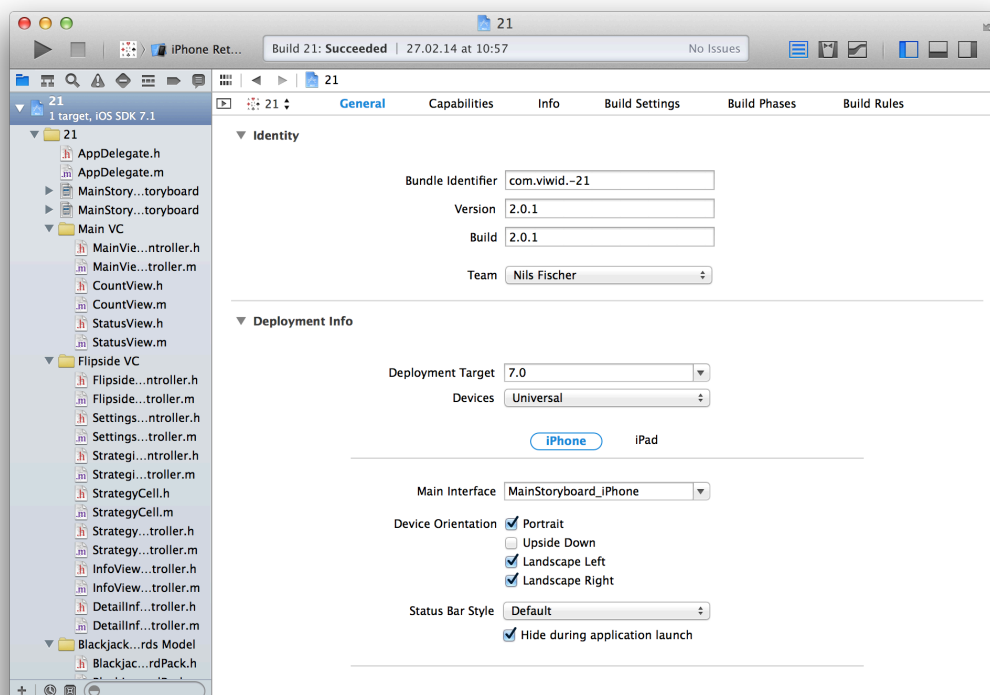


Abbildung 2.3: Die Projekt- und Targetkonfiguration wird angezeigt, wenn das Projekt links im Project Navigator ausgewählt ist

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.<deiname>.<productname>"` für Apps, die unserem Developer Team zugeordnet sind.

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 (s. S. 35, Abschnitt 4.1.2). In den meisten Fällen sollte hier die Storyboard-Datei 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. 12, Abb. 2.4).

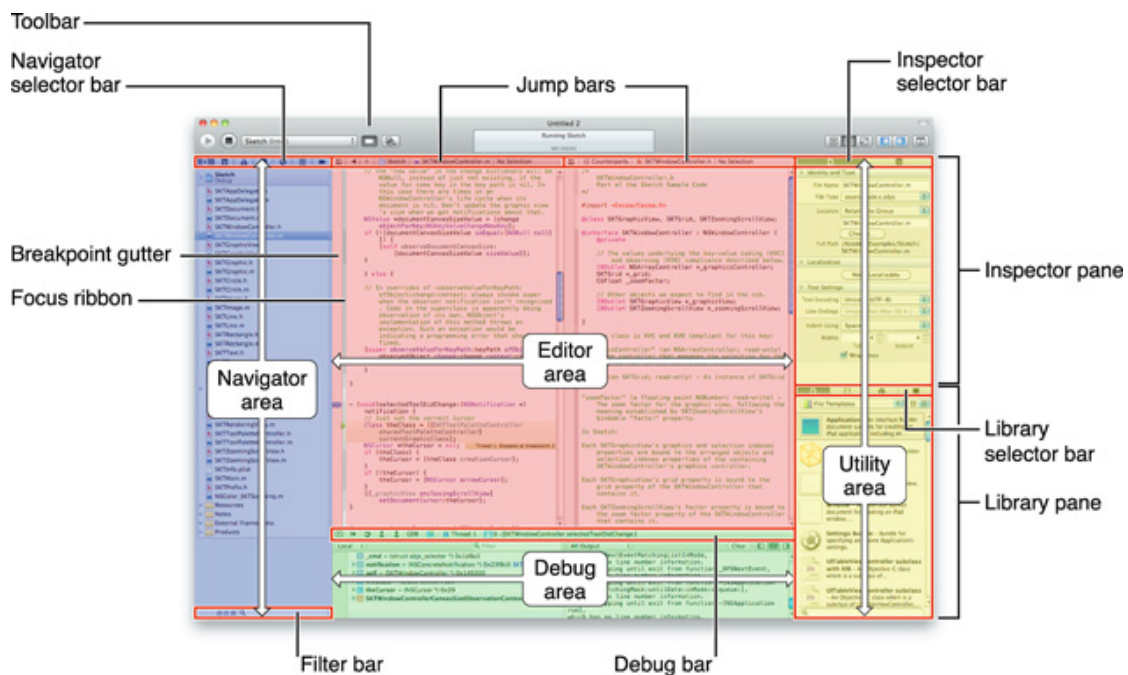


Abbildung 2.4: Xcode's Interface

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. Bezüglich Git und Versionskontrolle wird es später eine Einführung geben (s. S. 27, Abschnitt 3.1).


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



Es können ebenfalls mit  +  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  stehen noch weitere Optionen zur Verfügung. Da wir diese sehr häufig verwenden werden ist es sinnvoll, sich die Tastenkombinationen einzuprägen:

Build  +  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  +  Kompiliert das Target und führt das resultierende Product auf dem Zielsystem aus.

Stop  +  Beendet den aktiven Prozess.

Clean  +  +  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 gewählt werden, die über die Tastenkombinationen  +  bis  +  erreichbar sind:

1. **Project Navigator:** Übersicht über die Projektdateien
2. **Symbol Navigator:** Übersicht über alle Programmcodeelemente des Projektes
3. **Find Navigator:** Projektübergreifende Suche
4. **Issue Navigator:** Übersicht aller Warnung und Fehler des Compilers
5. **Test Navigator:** Übersicht aller erstellten automatischen Tests
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. 15, Abb. 2.5). 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.

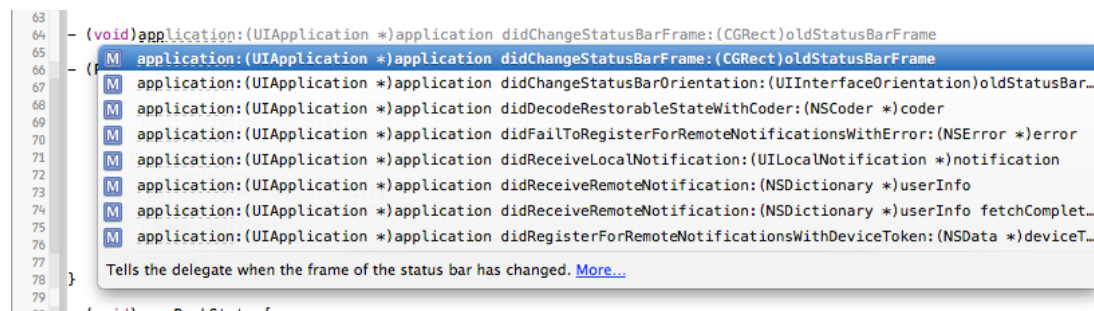


Abbildung 2.5: Der meiste Code wird mit der Autovervollständigung geschrieben

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

Swift ist eine sehr deskriptive Programmiersprache und Symbole in Apple's Frameworks sind nach Konventionen benannt. So können mit etwas Übung und Hilfe der Autovervollständigung auch unbekannte Symbole 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:

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. 2.6). Mit den Pfeiltasten können wir die gewünschte Option auswählen und mit der Enter-Taste einfügen.



Abbildung 2.6: 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. 16, Abb. 2.7). Dazu gehören fehlende Steuerzeichen wie Semikolons, aber auch komplexere Fehler.

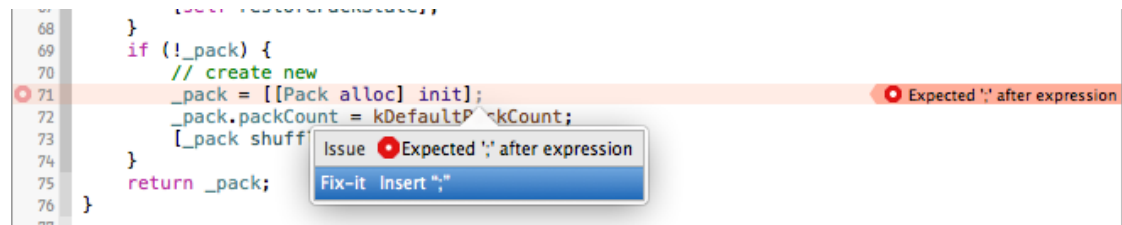


Abbildung 2.7: Xcode's Fehlerkorrektur erkennt und behebt Syntaxfehler

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. 2.8).

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.

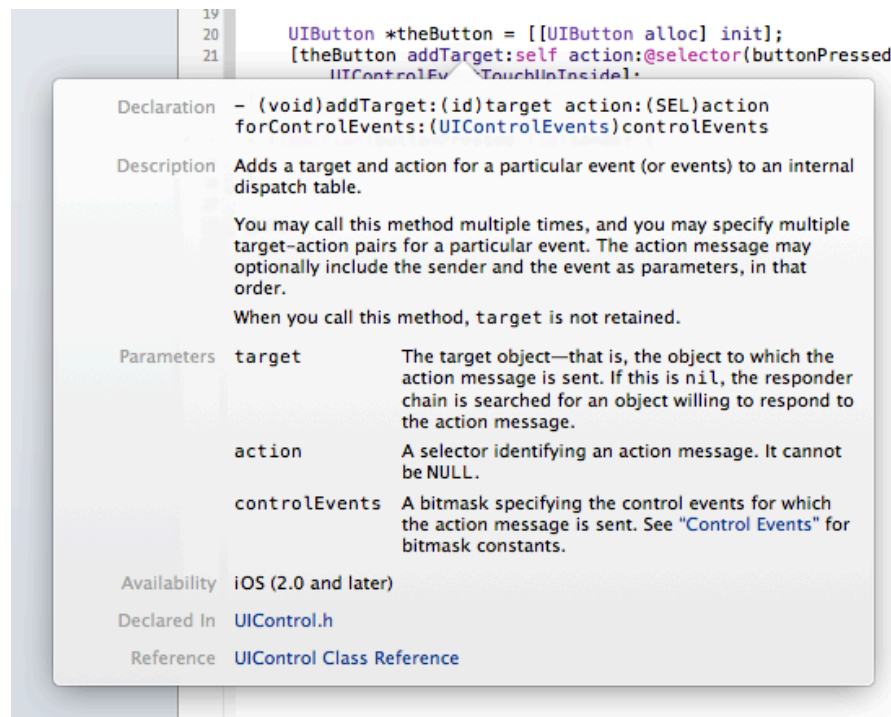


Abbildung 2.8: Alt-Klick auf ein Symbol zeigt eine kurze Definition

Tipp: Ist im Code an einer Stelle der Ausdruck `// MARK: Section Title` zu finden, so erscheint *Section Title* als Gliederung in dem Menü der Jump bar. Ein Minus erzeugt eine horizontale Line: `// MARK: – Section Title`.

Mit der *Open Quickly* Funktion `⌘+⇧+O` 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. Häufig werden wir den Assistent verwenden, um Interface und Code nebeneinander anzuzeigen und miteinander zu verbinden.

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. 18, 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 $\text{⌘} + \text{⇧} + 1$ bis $\text{⌘} + \text{⇧} + 6$ 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 $\text{⇧} + \text{⌘} + \text{H}$ - 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. 19, 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. 18, Abb. 2.9).

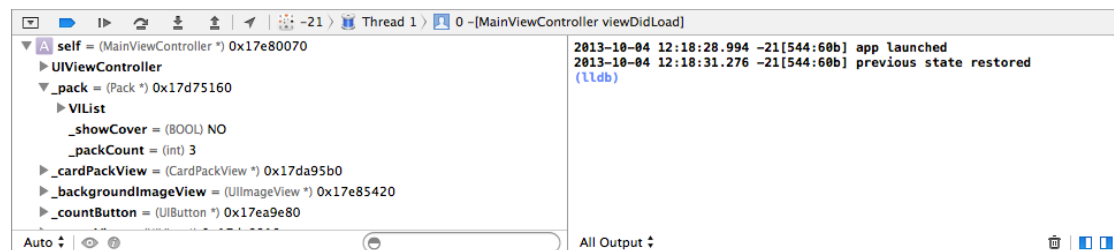


Abbildung 2.9: Im Debugger werden Konsolenausgaben und Situation der App angezeigt

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. 17, 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. Führt 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 das 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 Inspektor-Bereich ein (s. S. 20, Abb. 2.10). Im Editor-Bereich wird situationsbedingt der Standard- oder Assistant-Editor verwendet. Im IB-Modus wird im Editor-Bereich dann auf der 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

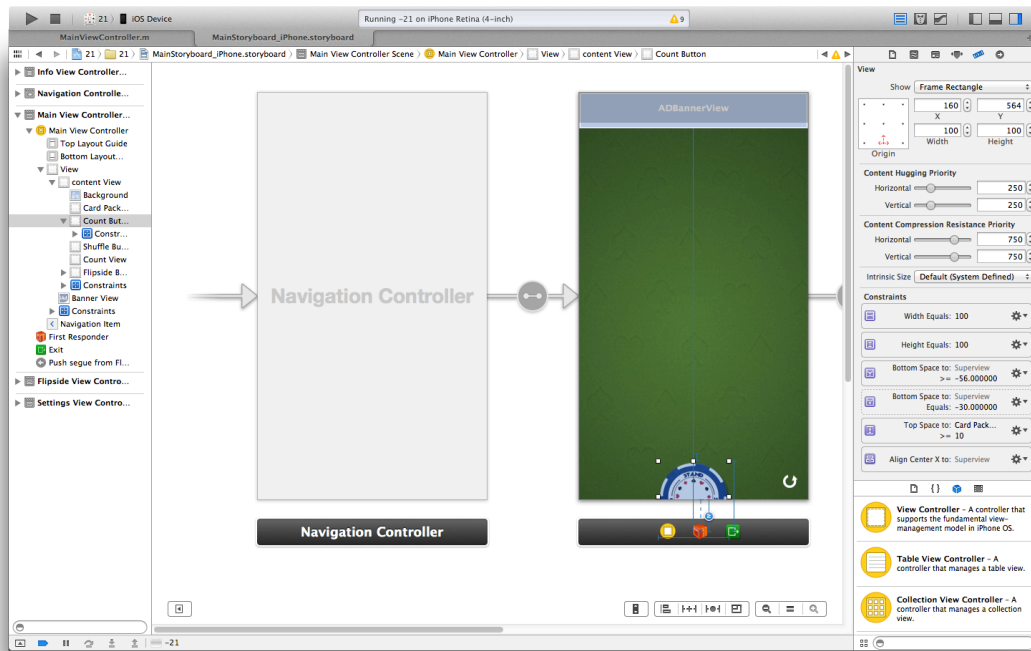


Abbildung 2.10: Um möglichst viel Platz zur UI-Gestaltung zu erhalten, verwenden wir für den Interface Builder eine angepasste Konfiguration

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.

2.3.1 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 und gesteuert wird.

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.

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. 21, Abschnitt 2.3.3).

2.3.3 IBOutlets & IBActions

Hinweis: Für diesen Abschnitt sind Kenntnisse über *Attribute* und *Methoden* der objekt-orientierten Programmierung 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 **Attribute** einer Klasse:

```
1 @IBOutlet var label: UILabel! // Dieses Attribut ist als IBOutlet
   ↳ gekennzeichnet.
```

In der XIB oder dem Storyboard können wir dann eine Verbindung zwischen dem Objekt und dem IBOutlet herstellen (s. S. 21, Abb. 2.11).

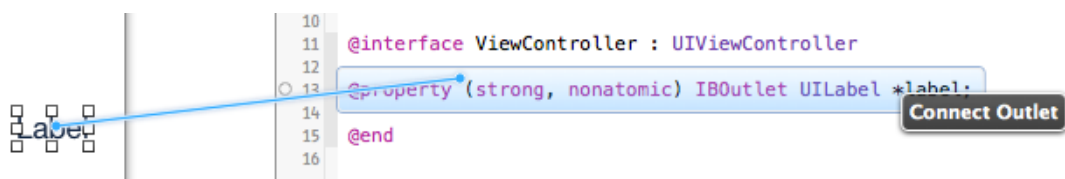



Abbildung 2.11: IBOutlets verbinden Interfaceelemente mit Properties im Code

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

```

1 // Sei theLabel das zuvor im Code erstellte Interface-Objekt
2 self.label = theLabel

```

Um sehr einfach IBOutlets zu erstellen, wechseln wir in den Assistant-Editor. Im Editor Bereich wird dann rechts der Assistant angezeigt. Hier können wir oben in der Jump bar  `Klassenname.swift` wählen, also die zugehörige Klasse, in der das Attribut deklariert wurde. Mit gedrückter `ctrl`-Taste können wir nun eine Verbindung zwischen dem Interfaceelement und dem Attribut ziehen (s. S. 21, Abb. 2.11).

Alternativ kann der Connection Inspector des Objekts (s. S. 21, 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 Dokumentübersicht links.

Als IBOutlets markierte Attribute werden häufig als **private** markiert, wenn nur innerhalb der Klasse auf die Interfaceelemente zugegriffen werden muss. Da sie außerdem meist automatisch geladen, mit dem Attribut verknüpft und dann nicht mehr entfernt werden, erhalten sie zusätzlich die Markierung **weak** und werden als *Implicitly Unwrapped Optional* deklariert:

```

1 @IBOutlet private weak var label: UILabel!

```

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. 2.12):

```

1 @IBAction func buttonPressed(sender: UIButton) { } // Diese Methode ist
   ↳ als IBAction gekennzeichnet.

```

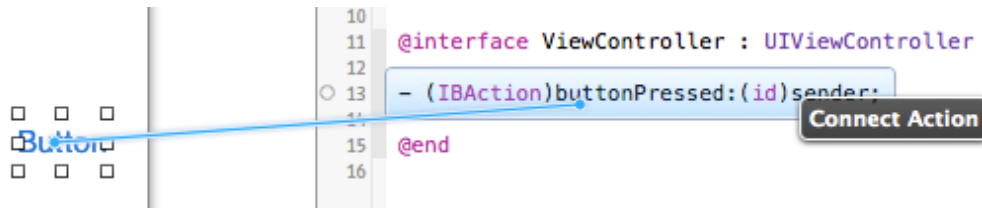


Abbildung 2.12: IBActions verbinden Events mit Methoden im Code

Die verbundene Methode wird dann ausgeführt, wenn das entsprechende Event ausgelöst wird. Das auslösende Objekt wird der Methode als Parameter sender übergeben.


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







```

1 // Sei theButton das das Event auslösende Objekt und theReceiver das
  ↳ Objekt, das die Methode buttonPressed: implementiert
2 theButton.addTarget(theReceiver, action: "buttonPressed:",
  ↳ forControlEvents: .TouchUpInside)

```

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. 16, Abschnitt 2.2.4). Ein Klick darauf öffnet den entsprechenden Abschnitt der Dokumentation im separaten Documentation-Fenster (s. S. 24, Abb. 2.13).

Dieses ist außerdem immer mit dem Tastenkürzel  +  +  (bzw.  +  +  +  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 ^[2] 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. 5, Abschnitt 1.3).

²<https://developer.apple.com/library/ios/>

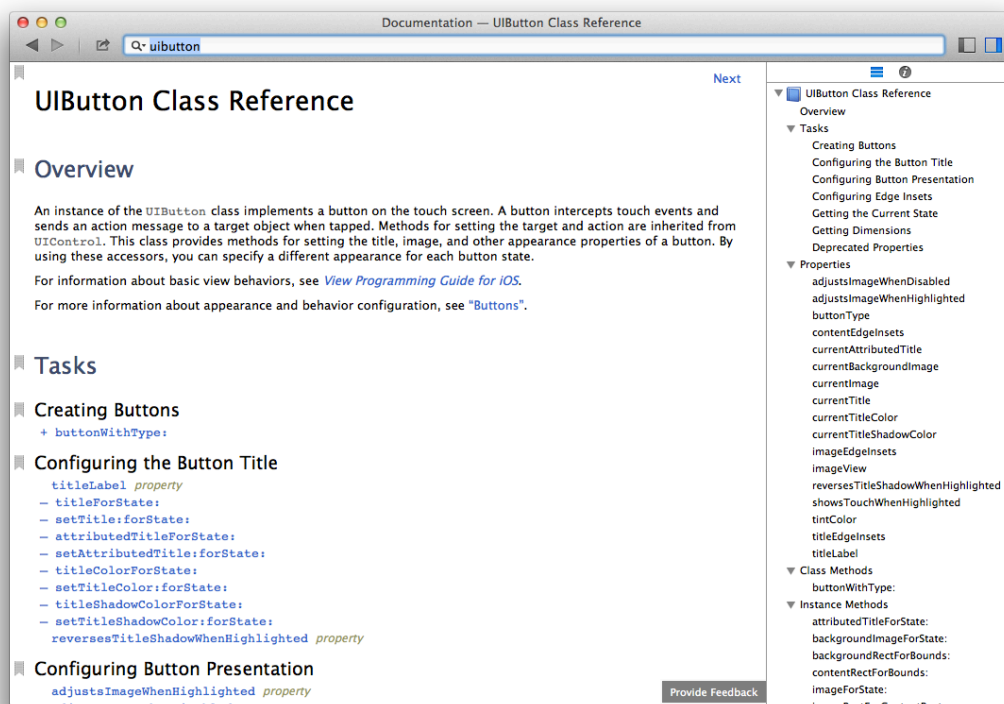


Abbildung 2.13: Xcode's integrierte Dokumentation ist beispielhaft strukturiert und sollte immer als erste Referenz verwendet werden

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. 25, Abb. 2.14).

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

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. 10, 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.

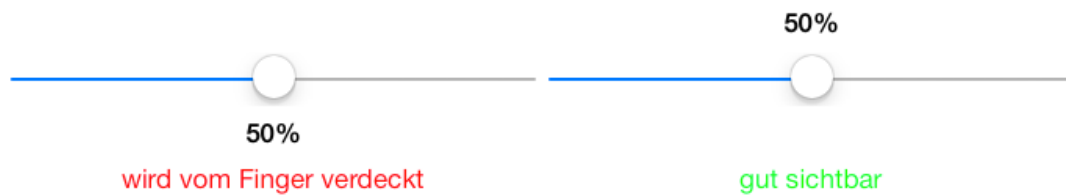


Abbildung 2.14: Unter einer Schaltfläche positionierte Interfaceelemente sind bei der Bedienung meistens vom Finger bedeckt. Solche Probleme werden häufig erst beim Testen auf realen iOS Geräten entdeckt.

2.5.1 Der Provisioning Prozess

Hinweis: Dieser Abschnitt stellt die Grundlagen des Provisioning Prozesses dar. Xcode vereinfacht diesen Vorgang mittlerweile sehr. Zum Testen von Apps auf iOS Geräten ist nicht einmal mehr ein Apple Developer Account notwendig, sodass dieser Abschnitt erst relevant wird, wenn ihr mit dem App Store arbeitet.

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 ^[3] 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

³<https://developer.apple.com/membercenter/>

ein Asterisk (*) 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.

Kapitel 3

Projektstrukturierung

3.1 Versionskontrolle mit Git

Die Entwicklung von iOS Apps ist wie jedes Programmierprojekt ein fortschreitender Prozess. Während an Features und Mechaniken im Code gearbeitet wird, möchte man häufig mal einen Ansatz ausprobieren oder Code umstrukturieren, doch wenn nötig schnell wieder zum letzten funktionierenden Stand zurückkehren können (s. S. 27, Abb. 3.1).





Abbildung 3.1: Mit Versionskontrolle können wir jederzeit speichern und dort wieder beginnen - nie wieder Game-Over!

Versionskontrolle (SCM / Software Configuration Management) bietet u.a. diese Sicherheit und ermöglicht es uns zusätzlich, einfach 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.

3.1.1 Grundlagen der Kommandozeilensyntax

Navigation `cd path/to/folder` navigiert zu dem angegebenen Pfad. Die Tilde `~` repräsentiert hier den Benutzerordner.  vervollständigt das aktuelle Pfadsegment und zweimaliges Drücken von  zeigt alle möglichen Vervollständigungen an.

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

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

3.1.2 Git Repository & Commits

Während wir an unserem Projekt arbeiten, verwenden wir Git, um Versionen unseres Codes 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 führen die Initialisierung durch:

```
1 cd path/to/project
2 git init
3 >> Initialized empty Git repository in path/to/project/.git/
4 git status
5 >> On branch master
6 >>
7 >> Initial commit
8 >>
9 >> nothing to commit (create/copy files and use "git add" to track)
```

`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. `git diff` listet die Änderungen sogar detailliert auf. Befindet sich der Code in einem akzeptablen Zwischenzustand, können wir die Änderungen mit einem Commit im Repository sichern:

```
1 git add --all # Markiert alle Änderungen für den nächsten Commit
   ↪ ("Staging")
2 git commit -m "Commit Message"
3 >> [master (root-commit) a74833f] Commit Message
4 >> x files changed, y insertions(+), z deletions(-)
```

```
5 git log
6 >> ...
```

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.

3.1.3 Branches

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

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.

Achtung: Da ein rebase die Commitfolge verändert, ist Vorsicht angebracht, wenn man mit anderen zusammenarbeitet!

```
1 git checkout master # Wechsle zurück in den Branch master
2 git merge new_feature # Führe alle Commits aus dem Branch new_feature mit
  ↳ dem aktiven Branch zusammen
```

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.

Wenn sich bspw. plötzlich ein User unserer App über einen Fehler beschwert, während wir bereits eifrig in einem Feature Branch an der nächsten Version arbeiten, können wir schnell zurück in den `master` Branch wechseln, den Fehler beheben und veröffentlichen. Zurück im Feature Branch kann diese Fehlerbehebung mit einem `merge` direkt integriert und dann weitergearbeitet werden.

3.1.4 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. 33, Abschnitt 3.1.7) enthält detaillierte Beschreibungen zu Remote Repositories.

An dieser Stelle darf der Service **GitHub**^[1] nicht unerwähnt bleiben, der Entwicklern eine Plattform für ihre Git Repositories bietet und für öffentliche Projekte kostenlos ist. Eine Alternative, die auch auf einem eigenen Server installiert werden kann, bietet bspw. **GitLab**^[2].

Ein Repository kann mit `git clone` von einem Server 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. In dieser Form sind Entwickler auf der ganzen Welt in der Lage, an Open Source Projekten zusammenzuarbeiten.

Dieser Kurs auf GitHub

Git ist für viele Projekte nützlich und bei weitem nicht nur auf Programmcode beschränkt. So befindet sich bspw. neben der Vorlesungswebseite auch dieses Skript in einem Git Repository auf GitHub ^[3]. Ihr könnt es also leicht klonen und Aktualisierungen herunterladen, wenn diese online verfügbar sind:

```
1 cd path/to/directory
2 git clone https://github.com/iOS-Dev-Kurs/Skript
3 # zur Aktualisierung:
4 git pull
```

Ihr seid außerdem herzlich dazu eingeladen, Verbesserungen als Pull Requests vorzuschlagen!

3.1.5 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 der *Version Editor* (s. S. 32, Abb. 3.2), 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

¹<http://www.github.com>

²<https://gitlab.com/>

³<https://github.com/iOS-Dev-Kurs>

angezeigt und Änderungen visualisiert. Mit der Uhr-Schaltfläche unten in der Mittel-leiste können frühere Commits ausgewählt werden.

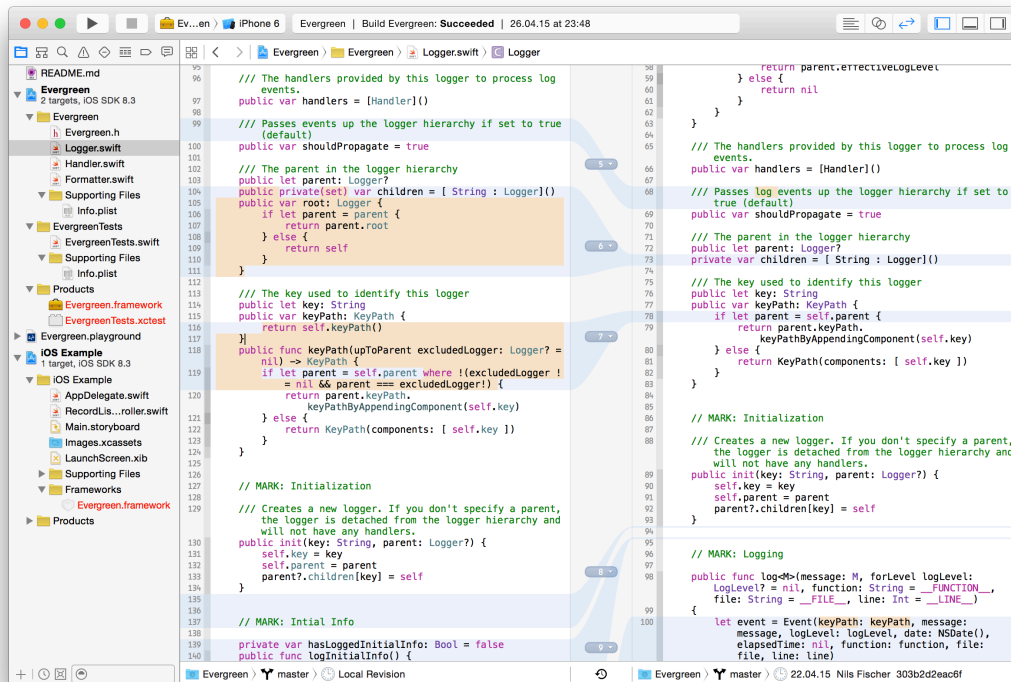


Abbildung 3.2: Der Version Editor zeigt die links geöffnete Datei zu einem früheren Commit an und visualisiert Änderungen

3.1.6 Gitignore

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

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

Mit einem Texteditor wie der TextEdit App oder vim in der Konsole können wir diese Datei nun editieren. Sinnvolle Vorlagen für verschiedenste Programmiersprachen und Plattformen sind im GitHub Repository *gitignore* [4] zu finden. Kopiert für Xcode Projekte die Vorlagen *Xcode* [5] und *OSX* [6] in die `.gitignore` Datei. Letztere Vorlage

⁴<https://github.com/github/gitignore>

⁵<https://github.com/github/gitignore/blob/master/Global/Xcode.gitignore>

⁶<https://github.com/github/gitignore/blob/master/Global/OSX.gitignore>

enthält bspw. die Zeile `.DS_Store`. So heißen versteckte Dateien in Mac OS X, die dem Betriebssystem verschiedene Informationen über den Ordner bereitstellen ^[7] und nicht in das Git Repository übernommen werden sollen.

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

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

3.1.7 Dokumentation

Git ^[8] ist erstklassig dokumentiert und bietet neben Tutorials und einem Übungsmodus ^[9] eine umfassende Dokumentation ^[10]. Hier sollte bei Bedarf unbedingt nachgeschlagen werden.

⁷http://en.wikipedia.org/wiki/.DS_Store

⁸<http://git-scm.com>

⁹<http://try.github.com/>

¹⁰<http://git-scm.com/book>

Kapitel 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 *Suspended* 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 und damit in den *Not running* State übergehen.

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

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 Funktion `main` in der Datei `main.swift` wird aufgerufen. Diese Funktion wird auch als *Entry Point* bezeichnet und besitzt entsprechende Äquivalente in anderen Programmiersprachen.

`main` ist dafür zuständig, die `UIApplicationMain` Funktion des `UIKit` Frameworks aufzurufen und ihr die Klassen des *Application Object* und *Application Delegate* zu übergeben (s.u.).

Da die Implementierung der `main` Funktion fast immer gleich ist, stellt das `UI`-Kit Framework stattdessen das Attribut `@UIApplicationMain` zur Verfügung, mit dem eine Klasse als *Application Delegate* markiert werden kann. Das *Application Object* ist dann vom Typ `UIApplication`.

```
1 @UIApplicationMain
2 class AppDelegate: UIResponder, UIApplicationDelegate {
3     var window: UIWindow?
4     // ...
5 }
```

3. Die `UIApplicationMain` Funktion erzeugt das **Application Object**, also ein Objekt der Klasse `UIApplication`, das für die zentrale Koordination der App verantwortlich ist.
4. Anschließend wird das **Application Delegate** als Objekt der zuvor mit dem `@UIApplicationMain` Attribut markierten Klasse erstellt und dem Attribut `delegate` des *Application Objects* zugewiesen.

Das *Application Delegate* stellt das Kernstück unseres selbstgeschriebenen Codes dar.

Während das *Application Object* die Ausführung unserer App koordiniert, bietet es dem *Application Delegate* im Verlauf des Startvorgangs und während der Laufzeit der App eine Vielzahl von Möglichkeiten, auf Ereignisse zu reagieren. Dazu werden Methoden des `UIApplicationDelegate` Protokolls aufgerufen. Diese informieren bspw. über den Wechsel der oben beschriebenen States und sind im Übersichtsdiagramm dargestellt (s. S. 37, Abb. 4.1).

Die wohl wichtigste Methode ist dabei `application:didFinishLaunchingWithOptions:`. Diese wird am Ende des Startvorgangs aufgerufen und zur initialen Konfiguration der App verwendet. Hier werden bspw. zentrale Datenstrukturen erstellt und das User Interface vorbereitet.

5. Im Xcode Projekt ist eine Datei `Info.plist` zu finden. Diese steht dem *Application Object* 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. Die im Storyboard enthaltenen Objekte werden dabei instanziiert und die Connections (IBOutlets und IBActions) hergestellt. Die so erstellte *View Hierarchie* (s. S. 39, Abschnitt 4.3) wird einem *UIWindow* Objekt hinzugefügt, das dann dem Attribut `window` des Application Delegate zugewiesen wird.

6. Nun übernimmt die *View Controller Hierarchie* (s. S. 46, Abschnitt 4.5) die Kontrolle über einzelne Ansichten unserer App, während das Application Delegate auf globale Ereignisse reagiert.

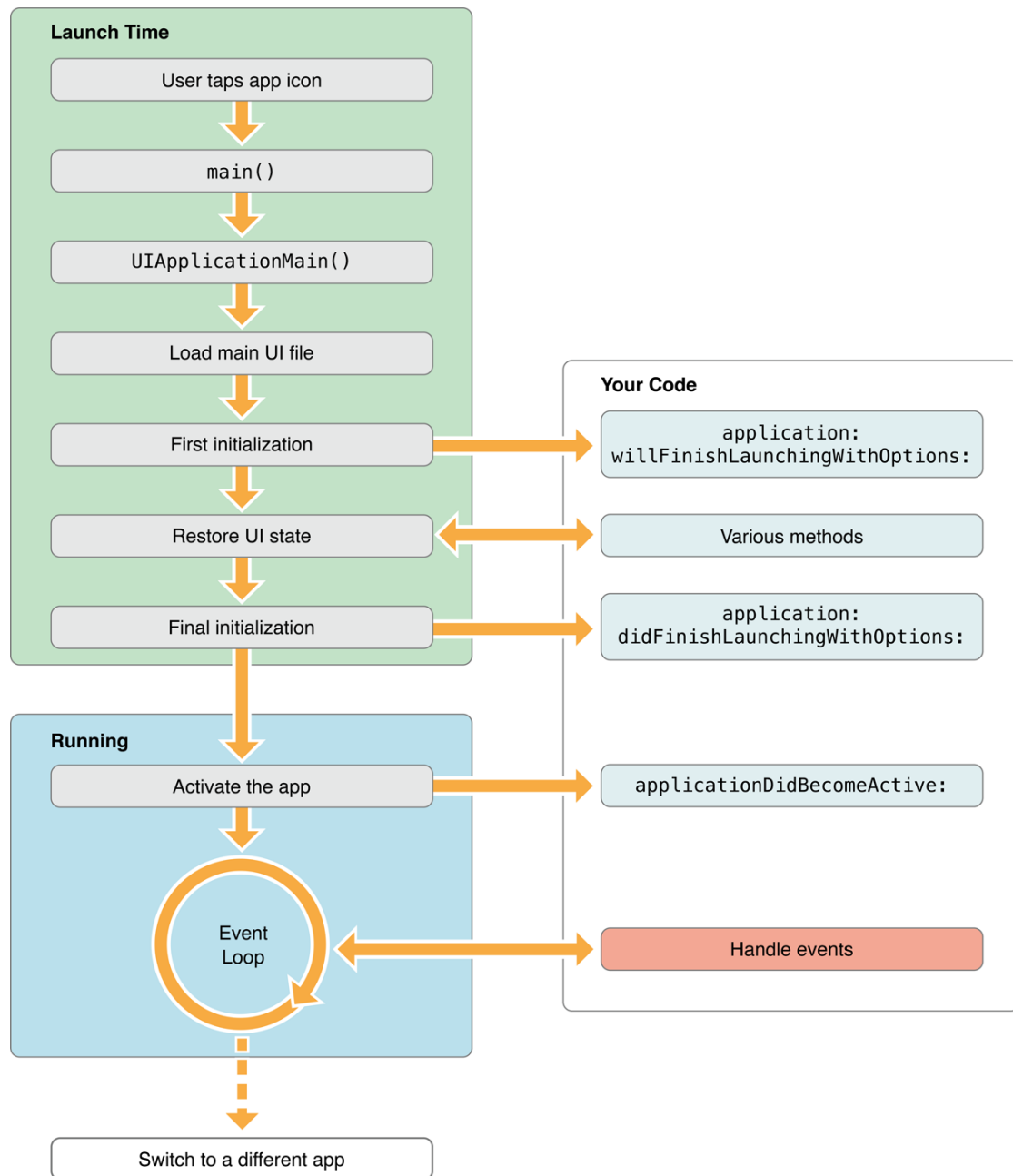


Abbildung 4.1: Übersicht der Prozesse beim Starten einer iOS App (aus dem iOS App Programming Guide)

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* eines Programms strikt getrennt implementiert werden (s. S. 38, Abb. 4.2). Durch diese Modularisierung bleibt ein Softwareprojekt flexibel und erweiterbar und einzelne Komponenten können leicht wiederverwertet werden.

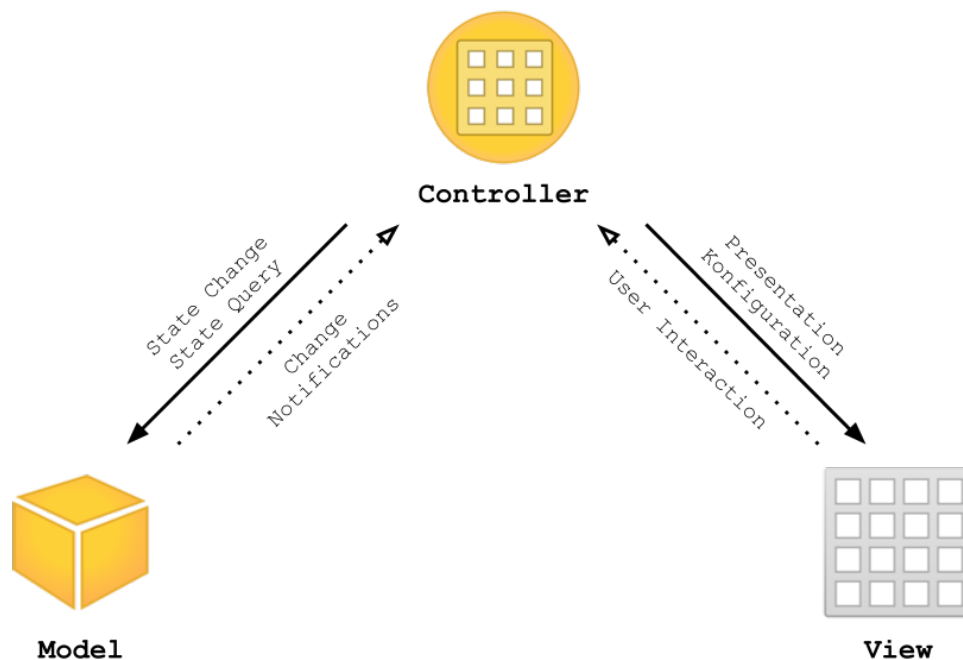


Abbildung 4.2: Das MVC Konzept trennt Modell, Präsentation und Steuerung

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

In Swift implementieren wir Klassen, Structs und Enumerations mit Attributen, Methoden und Beziehungen, um solche 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



View

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. 40, Abb. 4.3). An oberster Stelle der Hierarchie steht dabei ein Objekt der Klasse `UIWindow` : `UIView`, das von dem Application Object verwaltet wird (s. S. 35, 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 `frame` : `CGRect` 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. 40, Abb. 4.4) mit einer horizontalen x-Achse und vertikalen y-Achse in positiver Richtung nach rechts unten.

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

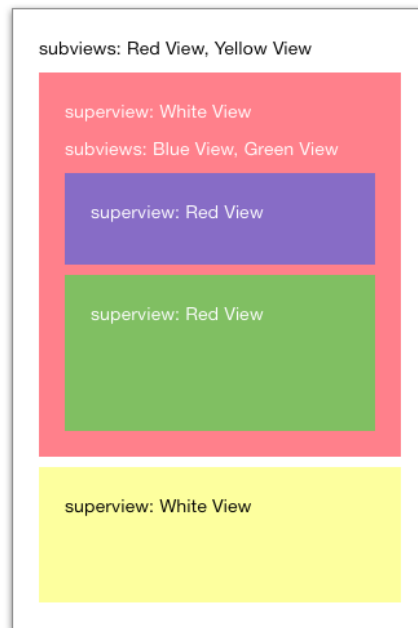


Abbildung 4.3: Jedes Objekt der `UIView` Klasse dient wieder als Container für weitere Objekte dieser Klasse

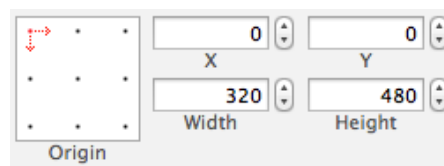


Abbildung 4.4: Der Ursprung des Koordinatensystems von `UIView` liegt in der oberen linken Ecke

```

1 struct CGRect {
2     var origin: CGPoint
3     var size: CGSize
4 }
5
6 struct CGPoint {
7     var x: CGFloat
8     var y: CGFloat
9 }
10
11 struct CGSize {
12     var width: CGFloat
13     var height: CGFloat
14 }

```

4.3.2 UIView Objekte

Wir können ein `UIView` Objekt u.a. mit dem Initializer `init(frame: CGRect)` erstellen und dann der View Hierarchie hinzufügen, sodass es auf dem Bildschirm angezeigt wird:

```
1 let view = UIView(frame: CGRect(x: 0, y: 0, width: 320, height: 44))
2 self.view.addSubview(view) // Angenommen, self.view ist bereits Teil der
   ↳ angezeigten View Hierarchie
```

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

4.4 Auto Layout



View

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 möglichst groß genug für ihren Inhalt sein, jedoch immer einen sinnvollen Mindestabstand von den Seiten des Bildschirms haben und den Inhalt wenn nötig komprimieren.
- Eine View soll wenn möglich zentriert positioniert sein, nie jedoch den Frame einer anderen View überlagern.
- Zwei Views sollen immer die gleiche Höhe besitzen, die sich an der benötigten Größe des Inhalts der jeweils größeren View orientiert.
- Eine View soll immer den gesamten Bereich ihrer Superview ausfüllen.

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

4.4.1 Constraints

Jede Constraint ist ein Objekt der Klasse `NSLayoutConstraint` und 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 u.a. 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, wird eine Warnung ausgegeben und die Superview hebt 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. Ein solches liegt vor, wenn jeder Subview ein eindeutiger Wert für alle vier Parameter des `frame` Attributs zugewiesen werden kann.

4.4.2 Intrinsic Content Size

Subklassen von `UIView` 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. 43, Abb. 4.5).

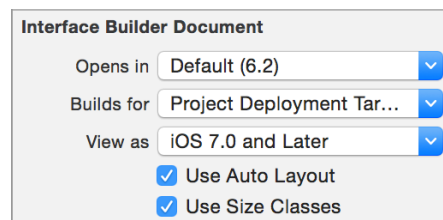


Abbildung 4.5: Auto Layout kann auf Basis einzelner Interface Builder Dateien ein- oder ausgeschaltet werden

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. 43, Abb. 4.6). Hier finden wir die interaktiven Menüs *Align* und *Pin*, mit denen den ausgewählten Views Constraints hinzugefügt werden können.

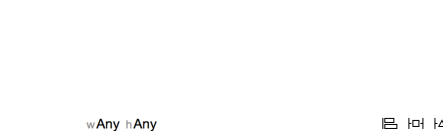


Abbildung 4.6: Die Schaltflächen am unteren rechten Rand des Editorbereichs bieten Zugriff auf viele Auto Layout Optionen

Alternativ kann das von IBOutlets und IBActions bekannte Ziehen einer Verbindungslinie bei gehaltener **ctrl**-Taste auch zum Erstellen von Constraints zwischen zwei Interfaceelementen verwendet werden (s. S. 44, Abb. 4.7). Dabei wird abhängig von der Richtung der 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. 44, Abb. 4.8). Solange das Layout für die View noch uneindeutig ist, werden

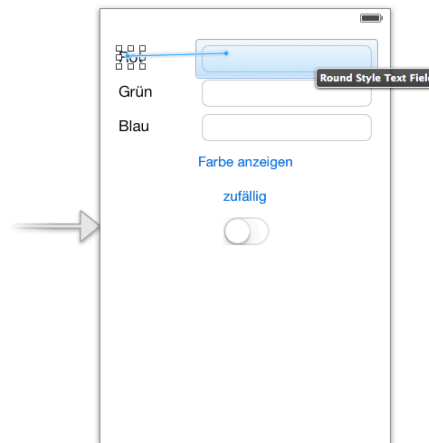


Abbildung 4.7: Constraints können ähnlich wie IBOutlet und IBAction durch Ziehen einer Verbindungslinie erstellt werden

die Constraints in gelb angezeigt. Eindeutige Layouts werden blau und Layouts mit Konflikten rot markiert.

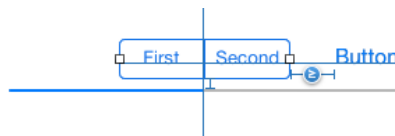


Abbildung 4.8: Ist das Layout einer View durch Constraints eindeutig festgelegt, werden diese blau gekennzeichnet

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. 44, Abb. 4.9). 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.



Abbildung 4.9: Uneindeutige Layouts oder Views, deren Position von vom berechneten Layout abweichen, werden gelb gekennzeichnet und zeigen ihre Position zur Laufzeit als eine gestrichelte Markierung

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. 45, Abb. 4.10).

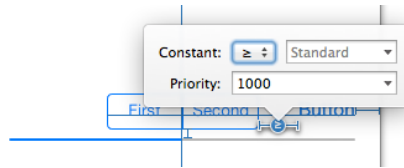


Abbildung 4.10: Ein Doppelklick auf eine Constraint im Interface Builder zeigt die wichtigsten Optionen in einem Popup

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 `constraints` Zugriff auf alle Constraints.

Einzelne Constraints können mit der Klassenmethode `constraintWithItem:attribute:relatedBy toItem:` instanziiert werden. Da jedoch meist mehrere Constraints benötigt werden, stellt `NSLayoutConstraint` zusätzlich eine Klassenmethode `constraintsWithVisualFormat:options:metrics:views` zur Verfügung, die die *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^[2] einsehbar.

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

```
1 let constraints = NSLayoutConstraint.constraintsWithVisualFormat("|-
  ↳ [view1]-10-[view2]-|", options: .allZeros, metrics: nil, views: [
  ↳ "view1" : self.view1, "view2" : self.view2 ])
2 self.view.addConstraints(constraints)
```

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

4.4.5 Size Classes

Mit Auto Layout werden Constraints zunächst für beliebige Displaygrößen definiert. Mit dem Konzept der *Size Classes* können dann einzelne Constraints oder das ganze Layout für bestimmte Displaygrößen angepasst werden.

Es gibt in horizontaler und vertikaler Richtung jeweils die Size Classes *Compact* und *Regular*. Jede Klasse, die das `UITraitEnvironment` Protokoll implementiert, also u.a. `UIView`, stellt unter dem Attribut `traitCollection` neben anderen Eigenschaften ihre `horizontalSizeClass` und `verticalSizeClass` zu Verfügung.

Size Classes werden in der View Hierarchie vererbt, beginnend beim `UIScreen` des ausführenden Geräts.

- Alle iPhones in Portrait Orientierung sind horizontal Compact und vertikal Regular.
- In Landscape Orientierung sind iPhones sowohl horizontal als auch vertikal Compact. Nur iPhones mit 5.5-inch Bildschirm sind horizontal Regular.
- iPads sind sowohl horizontal als auch vertikal Regular.

Weiterhin verändern einige View Controller Container (s. S. 50, Abschnitt 4.5.3) die Size Class ihrer View Hierarchie, wenn einer View nur ein Teil des Bildschirms zur Verfügung steht.

Wenn für eine Interface Builder Datei Size Classes aktiviert sind (s. S. 43, Abb. 4.5), kann am unteren Rand des Editorbereichs *Any* oder eine bestimmte Size Class ausgewählt werden (s. S. 47, Abb. 4.11). Das Layout wird zunächst für *Any* konfiguriert und dann für bestimmte Size Classes überschrieben, wobei Interfaceelemente und Constraints beliebig verändert, hinzugefügt oder entfernt werden können.

4.5 View Controller Hierarchie



Controller

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

Diese Subklassen von `UIViewController` sind der *Controller*-Komponente des Model-View-Controller Konzepts zugeordnet (s. S. 38, Abschnitt 4.2).

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

Zu den Aufgaben eines View Controllers gehören:

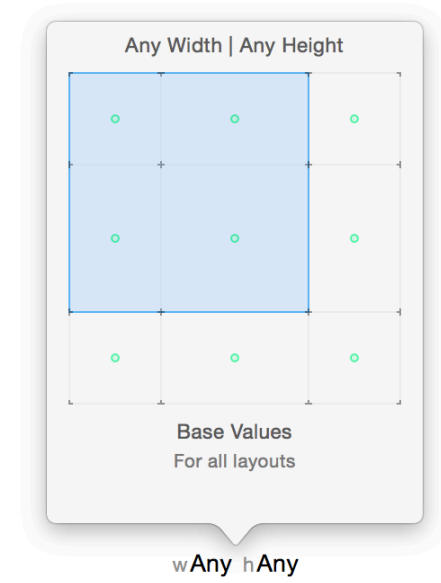


Abbildung 4.11: Am unteren Rand des Editorbereichs kann eine bestimmte Size Class ausgewählt werden, um das Layout für diese anzupassen.

- die dynamische Positionierung der Views in der View Hierarchie 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. 48, Abb. 4.12). Demnach gibt es übergeordnete View Controller, die die Content Views anderer View Controller der View Hierarchie ihrer eigenen Content View hinzufügen (s. S. 49, Abb. 4.13).

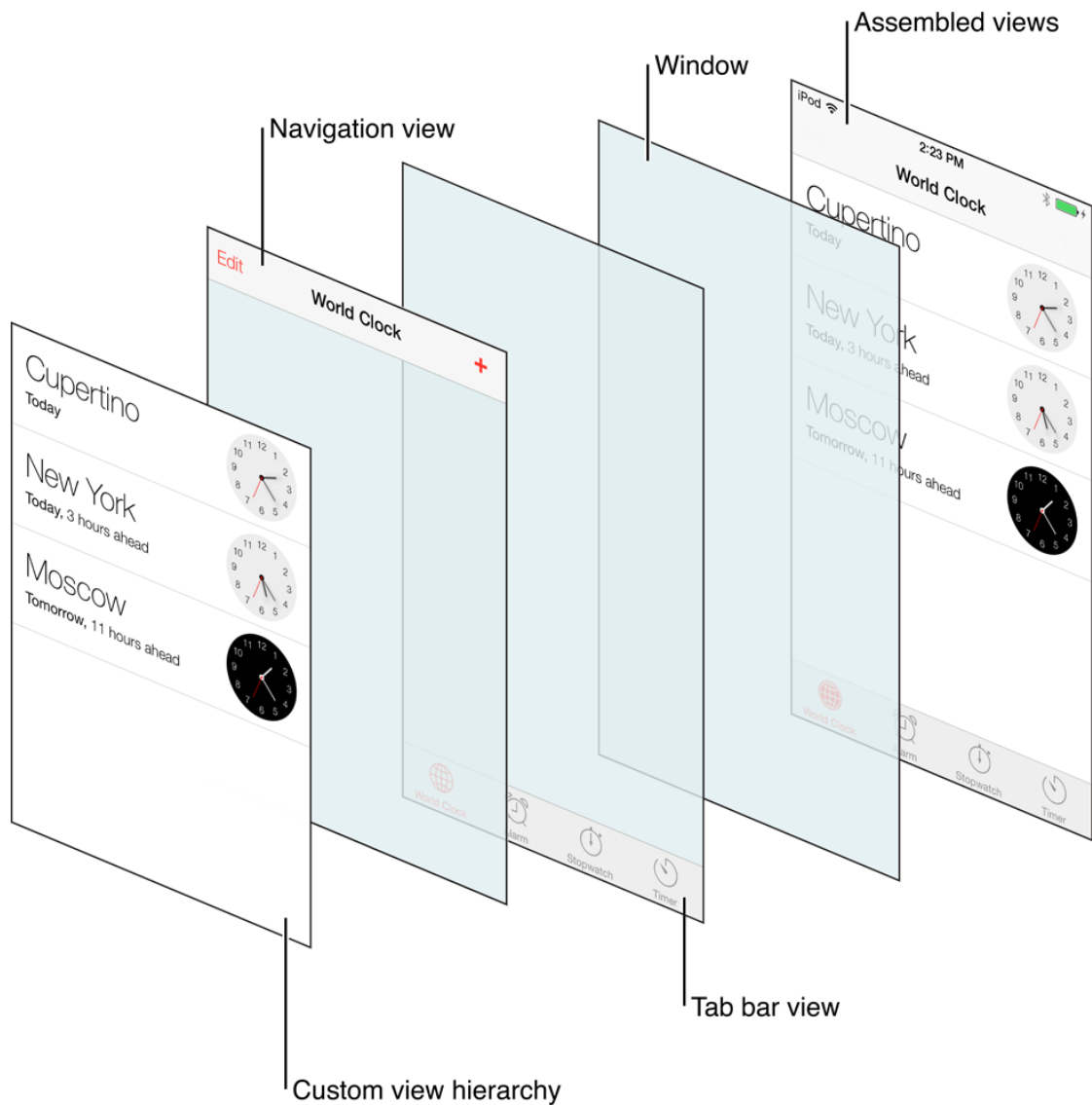


Abbildung 4.12: Container View Controller präsentieren Content Views anderer View Controller, sodass eine View Controller Hierarchy entsteht. Abbildung aus der UINavigationController Class Reference

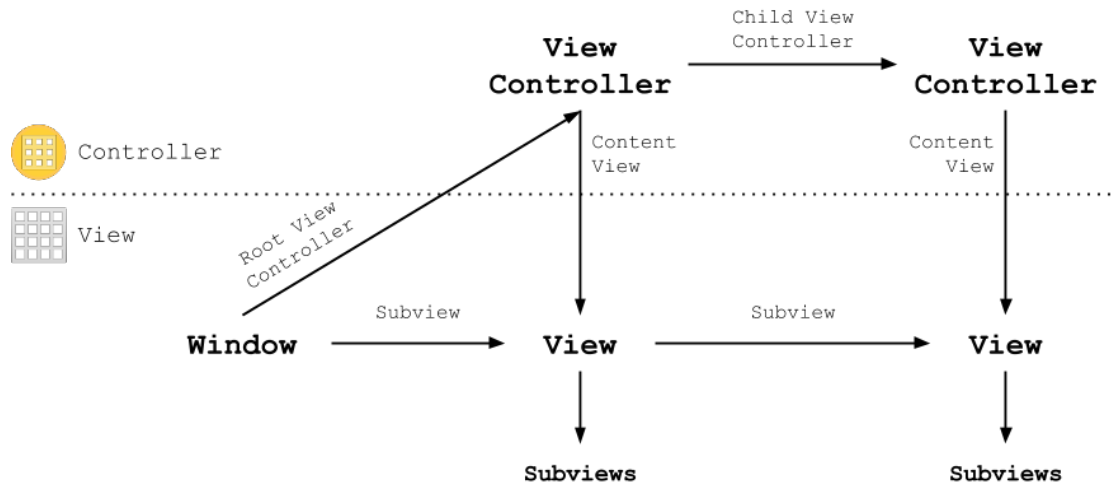


Abbildung 4.13: Jeder View Controller verwaltet seine Content View, die einen Teil der View Hierarchie darstellt

Die oberste Instanz der View Hierarchie ist ein Objekt der `UIWindow` Klasse. Analog besitzt `UIWindow` ein Attribut `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 müssen, wird man kaum eine iOS App ohne mindestens eine Subklasse von `UIViewController` finden. Häufig 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:`, `viewDidAppear:`, `viewWillDisappear:` 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: true,
   ↪ completion: nil)
3 // Beendet die Präsentation
4 self.dismissViewControllerAnimated(true, completion: nil)
```

UIKit implementiert darüber hinaus eine vielseitige API zur Präsentation von View Controllern. Dabei wird die View Controller Hierarchie traversiert und nach einem übergeordneten View Controller gesucht, der die Präsentation übernimmt. Die Container View Controller in UIKit sind sehr einfach zu verwenden (s. S. 50, Abschnitt 4.5.3)

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, bspw. in Form von Leisten. Stattdessen verwalten sie eine Hierarchie von weiteren View Controllern und präsentieren deren Content Views.

Navigation Controller

`UINavigationController` implementiert eine *Stack*^[3] Datenstruktur, die in Form eines Attributs `viewControllers` nach außen repräsentiert wird.

Einem Attribut `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.

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

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

```

1 // Präsentiere einen View Controller
2 self.navigationController.pushViewController(nextViewController,
    ↪ animated: true)
3 // Zurück zum vorherigen View Controller
4 self.navigationController.popViewControllerAnimated(true)

```

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 `navigationItem`: `UINavigationController`, das entsprechend konfiguriert werden kann und diese Informationen dem Navigation Controller bereitstellt.

```

1 self.navigationItem.title = "Titel"
2 self.title = "Titel" // äquivalente Abkürzung
3 self.navigationItem.rightBarButtonItem =
    ↪ UIBarButtonItem(barButtonItemSystemItem: .Done, target: self, action:
    ↪ "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. 51, Abb. 4.14). Zur Laufzeit wird der View Controller dann als Objekt dieser Subklasse instanziiert.

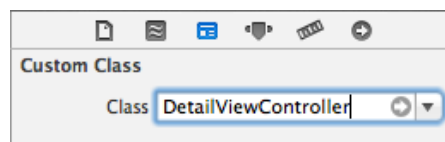


Abbildung 4.14: Im Identity Inspector kann bei ausgewählten View Controller dessen Subklasse ausgewählt werden

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

4.5.5 Segues

Zwischen Scenes vermitteln *Segues*. Diese stellen Beziehungen zwischen View Controllern dar und können zu deren Präsentation verwendet werden (s. S. 52, Abb. 4.15).

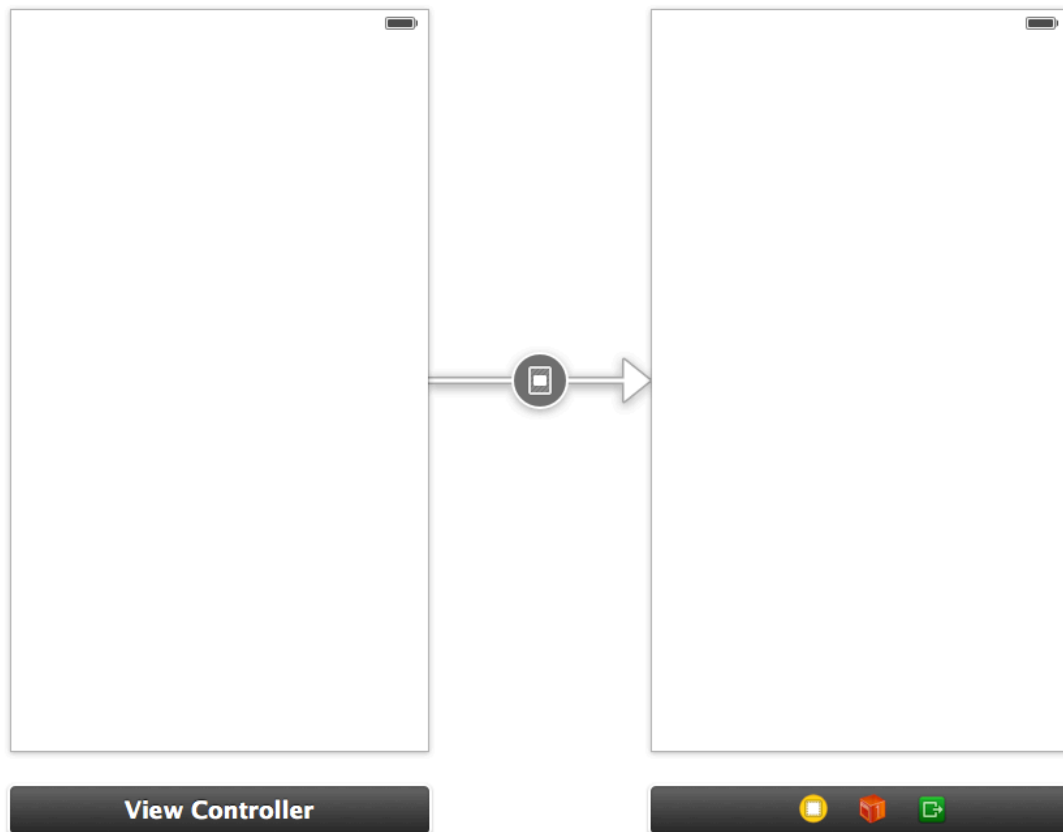


Abbildung 4.15: Segues vermitteln zwischen Scenes

Ähnlich wie bei IBOutlets, IBActions und Auto Layout Constraints gibt es mehrere Möglichkeiten, Segues im Interface Builder zu erstellen. Dazu gehören bspw. das Ziehen einer Verbindungslinie 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 *Relationship Segue* 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 `UIControlEvent` 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.
- Außerdem kann jeder View Controller *Unwind Segues* zur Verfügung stellen, die innerhalb dessen View Controller Hierarchie verwendet werden können, um zu diesem zurückzukehren.

Dazu ist lediglich die (ggf. leere) Implementierung einer `IBAction` Methode mit genau einem Argument vom Typ `UIStoryboardSegue` im Interface des View Controllers erforderlich. Der Methodenname sollte das Ziel der Unwind Segue beschreiben, also den implementierenden View Controller.

```
1 @IBAction func unwindToXYZ(segue: UIStoryboardSegue) { ... }
```

Die Unwind Segue ist anschließend im Storyboard in jedem View Controller in der Hierarchie des implementierenden View Controllers unter der *Exit* Schaltfläche verfügbar und kann mit einem `UIControlEvent` verbunden werden (s. S. 53, Abb. 4.16).



Abbildung 4.16: Die *Exit* Schaltfläche stellt alle Unwind Segues in der View Controller Hierarchie zur Verfügung

Im Attributes Inspector können Optionen bezüglich Übergangsanimation und Präsentationsmodus verschiedener Segues 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 wie im folgenden Beispiel:

```
1 override func prepareForSegue(segue: UIStoryboardSegue, sender:
  ↳ AnyObject?) {
2 {
3     if let identifier = segue.identifier {
4         switch identifier {
5             case "showDetailSegue":
6                 if let detailViewController = segue.destinationViewController
  ↳ as? DetailViewController {
```

```

7         detailViewController.detailObject = ...
8     }
9     default:
10        break
11    }
12 }
13 }

```

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 viel verwendeter Mechanismus der **einseitigen** Kommunikation zwischen Klassen, das einem einfachen **Frage-Antwort-Prinzip** folgt (s. S. 54, Abb. 4.17).



Abbildung 4.17: Das Delegate Konzept funktioniert nach einem einfachen Frage-Antwort-Prinzip

Die aktive (fragende) Klasse stellt ein *Protokoll* zur Verfügung, das die benötigten Methoden definiert. Außerdem besitzt sie ein öffentliches Attribut, 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. Die aktive Klasse muss dabei keine weiteren Informationen über die passive Klasse haben, außer, dass diese die benötigten Methoden des Protokolls implementiert.

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

- Von dem Delegate Objekt werden Informationen angefordert
- 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. Protokolle

sind tatsächlich so tief in Swift integriert, dass Swift auch die erste *protokollorientierte* (statt *objektorientierte*) Programmiersprache genannt wird.

Betrachten wir dazu das Beispiel eines View Controllers `SelectionViewController: UIViewController`, der präsentiert wird und zur Auswahl eines bestimmten Objekts aus einer Liste dienen soll. Der `SelectionViewController` erhält dafür ein Attribut `delegate`, das die Bedingungen eines `SelectionDelegate` Protokolls erfüllen muss. Dazu gehört eine Methode, die alle Objekte zurückgibt, aus denen ausgewählt werden kann. Der `SelectionViewController` kann diese dann darstellen.

Sobald eine Auswahl getroffen wurde, muss diese dem präsentierenden View Controller mitgeteilt werden. Das `SelectionDelegate` definiert dazu eine weitere Methode, die das Delegate implementieren muss und bei einer Auswahl aufgerufen wird. Das Delegate kann sich dann darum kümmern, den `SelectionViewController` wieder zu entfernen und mit dem ausgewählten Objekt weiterarbeiten.

Aktive Klasse

```
1 // SelectionViewController.swift
2 import UIKit
3
4 protocol SelectionDelegate: class {
5     func objectsForSelectionViewController(selectionVC:
6         ↪ SelectionViewController) -> [Object]
7     func selectionViewController(selectionVC: SelectionViewController,
8         ↪ didSelectObject object: Object)
9 }
10
11 class SelectionViewController: UIViewController {
12     weak var delegate: SelectionDelegate?
13
14     override func viewDidLoad() {
15         super.viewDidLoad()
16         let objects =
17             ↪ self.delegate?.objectsForSelectionViewController(self)
18         // configure view to display objects ...
19     }
20
21     // assume this method is called when an object is selected
22     func didSelectObject(object: Object) {
23         self.delegate?.selectionViewController(self, didSelectObject:
24             ↪ object)
25     }
26 }
```

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. Diese Art von Speicherverwaltung trifft nur auf Klassen, nicht auf Structs und Enums zu, weshalb das Protokoll mit **class** auf Klassen limitiert wird.

Passive Klasse

Wenn ein View Controller nun einen solchen `SelectionViewController` präsentiert, muss er ein `Delegate` zur Verfügung stellen, das die Methoden des `SelectionDelegate` Protokolls implementiert. Dies kann in viele Fällen der View Controller selbst übernehmen. Neben seiner Superklasse markiert er also, dass er das Protokoll erfüllt und implementiert die benötigten Methoden.

```
1 // ViewController.m
2 class ViewController: UIViewController, SelectionDelegate {
3
4     func showSelectionViewController() {
5         let selectionViewController = SelectionViewController()
6         selectionViewController.delegate = self
7         self.presentViewController(selectionViewController, animated:
8             ↪ true, completion: nil)
9     }
10
11     func objectsForSelectionViewController(selectionVC:
12         ↪ SelectionViewController) -> [Object] {
13         // obtain objects array
14         return objects
15     }
16
17     func selectionViewController(selectionViewController:
18         ↪ SelectionViewController, didSelectObject object: Object) {
19         // do something with the selected object ...
20         selectionViewController.dismissViewControllerAnimated(true,
21             ↪ completion: nil)
22     }
23 }
```

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 davon, 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 `section: Int` und `row: Int` besitzen.

Es kann zwischen den Darstellungsoptionen *Plain* und *Grouped* für Table Views gewählt werden, die lediglich das Erscheinungsbild der Table View verändern (s. S. 57, Abb. 4.18).

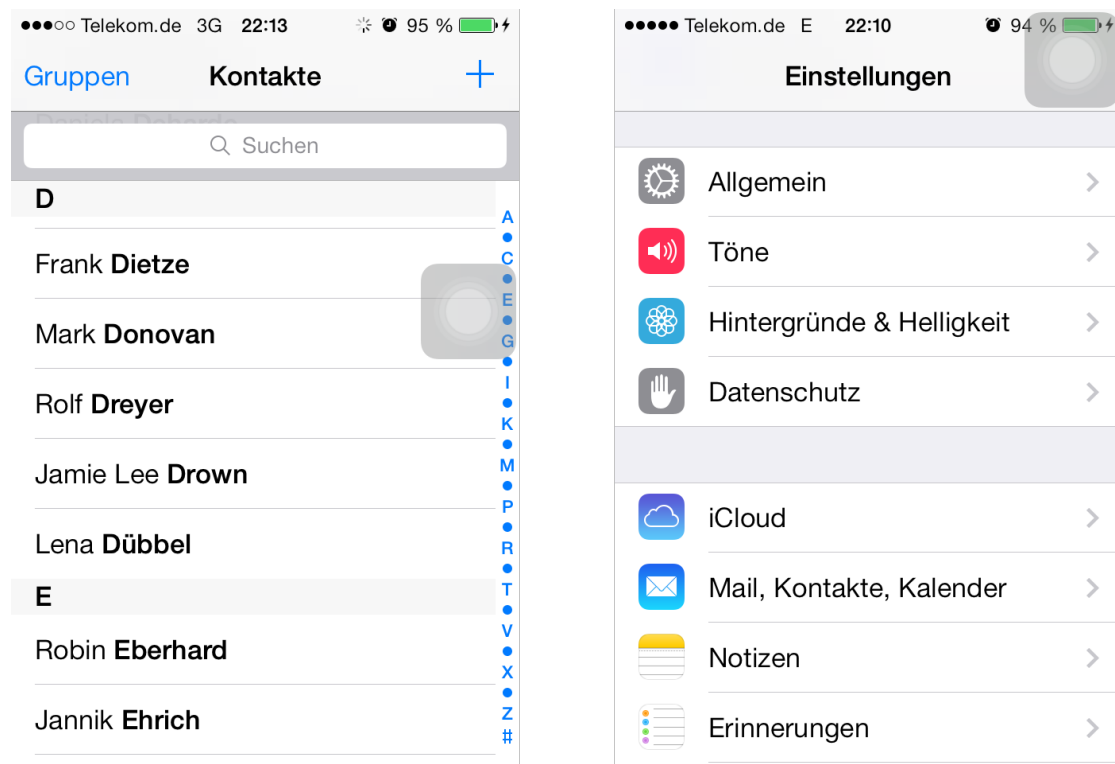


Abbildung 4.18: Table Views können in den Darstellungsoptionen Plain oder Grouped angezeigt werden

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 `datasource: UITableViewDataSource` und `delegate: UITableViewDelegate`. Zur Laufzeit 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. 58, Abb. 4.19). Natürlich eignet sich auch an dieser Stelle Auto Layout zur Positionierung der Views. Außerdem sollte jeder Prototype Cell ein Identifier zugewiesen werden.

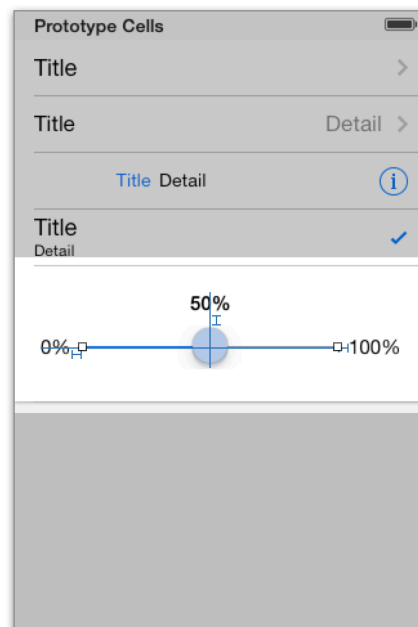


Abbildung 4.19: Prototype Cells können im Storyboard konfiguriert werden und dienen anschließend als Vorlage für dynamische Table View Cells

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 Hierarchie 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 class 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 `view: UIView` mit `tableView: UITableView` 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 gewohnt ausführlicher Form.

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

```
1 // Sei objects: [Object] ein Attribut dieser Subklasse
2 func numberOfSectionsInTableView(tableView: UITableView) -> Int {
3     // Bestimmt die Anzahl der Sections
4     return 1
5 }
6 func tableView(tableView: UITableView, numberOfRowsInSection section:
   ↪ Int) -> Int {
7     // Bestimmt die Anzahl der Rows für die angegebene Section
8     return objects.count
9 }
10 func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath:
   ↪ NSIndexPath) -> UITableViewCell {
11     // Greift auf das Model zurück, um das darzustellende Objekt zu
   ↪ erhalten
12     let object = objects[indexPath.row]
```

```
13 // Erstellt ein neues UITableViewCell Objekt entsprechend der im
    ↳ Storyboard definierten Prototype Cell mit dem angegebenen
    ↳ Identifier oder verwendet eine existierende, momentan nicht
    ↳ verwendete Zelle mit diesem Identifier
14 let cell = tableView.dequeueReusableCellWithIdentifier("objectCell",
    ↳ forIndexPath: indexPath) as! UITableViewCell
15 // Konfiguriert die View entsprechend des Models
16 cell.textLabel?.text = object.description
17 }
```

An dieser Stelle wird das Model-View-Controller Konzept (s. S. 38, Abschnitt 4.2) 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 Darstellungsoptionen die Möglichkeit zur Reaktion auf verschiedene Ereignisse. Dazu gehört die vierte wichtige Methode der Table View Programmierung:

```
1 func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath:
    ↳ NSIndexPath) {
2     // do something...
3 }
```

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 override func prepareForSegue(segue: UIStoryboardSegue, sender:
    ↳ AnyObject?) {
2     let indexPath = self.tableView.indexPathForSelectedRow()
3     // get model object and configure destination view controller ...
4
5 }
```

4.8 Data Persistence

Aufgrund der Natur des Multitaskings auf der iOS Plattform kann der Ausführungsstatus einer App (s. S. 34, 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 eine oder mehrere der folgenden 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 // speichern
2 NSUserDefaults.standardUserDefaults().setObject("Alice",
    ↪ forKey:"user_name")
3 // auslesen
4 let userName =
    ↪ NSUserDefaults.standardUserDefaults().objectForKey("user_name")
```

Es können nur Objekte der Klassen `NSString`, `NSNumber`, `NSDate` und `NSData` oder ausschließlich mit solchen Objekten gefüllte Instanzen von `NSArray` und `NSDictionary` (bzw. ihre Swift-Äquivalente) in dieser Form gespeichert werden. Andere Klassen müssen daher zunächst zu `NSData` Objekten serialisiert werden. Wird dies notwendig, sollte jedoch bereits über die Integration von Core Data (s. S. 62, Abschnitt 4.8.2) nachgedacht werden.

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

```
1 // Sei image: UIImage hier ein Bildobjekt
2 let imageData = UIImagePNGRepresentation(image);
3 NSUserDefaults.standardUserDefaults().setObject(imageData,
    ↪ forKey:"image_data") // speichern
4 if let imageData =
    ↪ NSUserDefaults.standardUserDefaults().objectForKey("image_data") as?
    ↪ NSData { // auslesen
```

```

5     let image = UIImage(data: imageData)
6 }

```

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 dar.

Xcode stellt hier in Analogie zum Interface Builder einen Editor zur Verfügung, mit dem zunächst eine Datenstruktur konfiguriert werden kann (s. S. 62, Abb. 4.20). Anschließend werden Subklassen von `NSManagedObject`: `NSObject` verwendet, die Elemente der Datenstruktur als Klassen repräsentieren und von der Datenbank verwaltet werden.

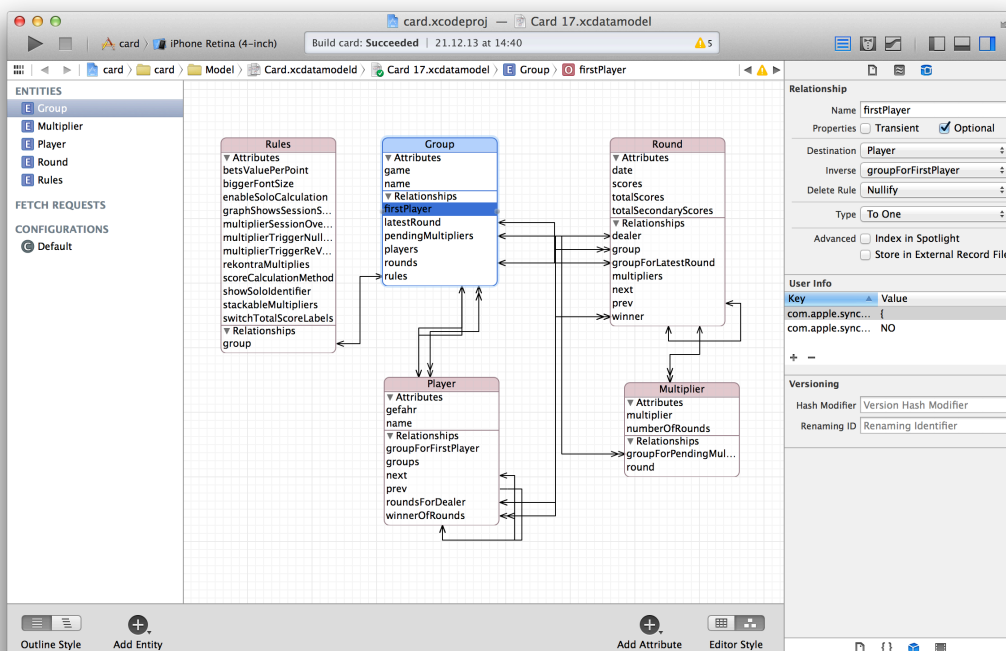


Abbildung 4.20: Core Data Models können in Analogie zum Interface Builder in Xcode graphisch bearbeitet werden

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 Apple's Entwicklern auf der WWDC vorgestellt (s. S. 5, Abschnitt 1.3). 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 der 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 `restorationIdentifier: String` 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. 54, 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. 61, Abschnitt 4.8.1) zu reagieren:

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

⁵<https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/>

```
1 // an beliebiger Stelle im Code, bspw. in der viewDidLoad Methode eines
   ↳ View Controllers
2 NotificationCenter.defaultCenter().addObserver(self, selector:
   ↳ "userDefaultsDidChange:", name: NSUserDefaultsDidChangeNotification,
   ↳ object: nil)
3
4 // in der Implementierung des View Controllers
5 func userDefaultsDidChange(notification: NSNotification) {
6     // update user interface ...
7 }
```

Kapitel 5



Entwicklungsprozess von iOS Apps

5.1 Unit Tests & Test Driven Development

Während der Softwareentwicklung werden häufig Umstrukturierungen vorgenommen, Features ergänzt oder verändert und Schnittstellen angepasst. Die Orientierung an grundlegenden Konzepten der Softwarestrukturierung wie dem Model-View-Controller Konzept vereinfacht solche Codeänderungen natürlich, doch es besteht weiterhin immer die Gefahr, dass zuvor funktionierender Code plötzlich Fehler aufwirft, die schwierig zu finden und zu beheben sein können.

Ein sehr sinnvolles Instrument zur Sicherung stabiler Software sind *Unit Tests*. Dies sind Methoden, die einzelne Codeblöcke (*units of code*) testen und immer entweder mit *Fail* oder *Pass* abschließen. Ein solcher Codeblock bezeichnet dabei *die kleinste testbare Einheit*. Häufig sind dies einzelne Methoden im Model, deren Rückgabewert kontextbedingt überprüft wird.

Einmal geschrieben werden alle Unit Tests häufig ausgeführt, bspw. periodisch auf einem Server oder bei Gelegenheit lokal. Treten dabei Probleme auf, sind die Ursachen bei gut geschriebenen Tests leicht zu finden. Häufig können so sogar Fehler gefunden werden, die bei normalem "User testing" übersehen werden.

Meist sind die Testmethoden in Subklassen einer bestimmten Klasse des entsprechenden Unit Test Frameworks organisiert. Auf iOS steht dazu das XCTest Framework [1] mit der Klasse XCTestCase zur Verfügung. Jede Testmethode ist durch den Prefix `test` des Methodennamens gekennzeichnet und erscheint damit bspw. automatisch im Xcode *Test Navigator* (s. S. 14, Abschnitt 2.2.3) und wird bei Betätigung von + automatisch ausgeführt. Außerdem werden die beiden Methoden `setUp` und `tearDown` der entsprechenden XCTestCase Subklasse vor bzw. nach dem Test ausgeführt.

Ein Beispiel für einen Unit Test ist folgende Methode in einer XCTestCase Subklasse:

¹https://developer.apple.com/library/ios/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UnitTestYourApp/UnitTestYourApp.html

```
1 func testEqualityOfDifferentStringsWithEqualContent() {  
2     let string = "content"  
3     let otherString = "content"  
4     XCTAssertEqual(string, otherString, "Strings with equal content are  
        ↪ not equal.")  
5 }
```

Die Swift Standard Library sollte diesen Test stets erfüllen, da `String` das `Equatable` Protokoll als Vergleich der enthaltenen Zeichenketten implementiert. Apple könnte diesen Test implementieren, sodass dieses Verhalten bei Codeänderungen immer überprüft und abgesichert wird.

Xcode 7 führt zusätzlich das XCUITest Framework für *UI Testing* ein. Dabei wird die Accessibility API verwendet, die zur alternativen Bedienung von Apps für Menschen mit Behinderungen konzipiert wurde, um das User Interface im Code zu steuern. Dann kann der Zustand der App an entsprechenden Stellen überprüft werden. Dabei werden User Interface Elemente durch *Proxy* Objekte repräsentiert, die durch ihr Accessibility Label gekennzeichnet sind. Im folgenden Beispiel wird ein Tap auf einen Button simuliert und anschließend überprüft, ob dadurch ein Gruß angezeigt wurde:

```
1 let app = XCUIApplication()  
2 XCTAssertFalse(app.staticText["Hello World!"].exists, "Greeting shown  
    ↪ before button was pressed")  
3 app.buttons["Say Hi"].tap()  
4 XCTAssertTrue(app.staticText["Hello World!"].exists, "Greeting not  
    ↪ shown upon button press")
```

Unit Tests sollten Teil jedes größeren Softwareprojekts sein. Einige Entwickler gehen sogar noch einen Schritt weiter und schreiben nach dem Prinzip des *Test Driven Development (TDD)* [2] bei jedem Abschnitt des Entwicklungsprozesses zuallererst die Unit Tests, die der Code letztendlich erfüllen soll und die die gewünschte Funktionalität definieren. Dann wird daran gearbeitet, diese Tests möglichst einfach zu erfüllen, bevor die nächsten Unit Tests definiert werden. Diese Vorgehensweise führt zu Code, bei dem idealerweise kein aufwändiges Debugging betrieben werden muss, da er aus einzelnen, umfassend getesteten Segmenten besteht.

5.2 Open Source Libraries & CocoaPods

Zur Entwicklung unserer Apps haben wir bisher die von Apple bereitgestellten Frameworks wie Foundation und UIKit verwendet. Diese bilden die Grundlage jeder iOS App, doch natürlich stoßen wir im Verlauf der Entwicklung einer oder mehrerer Apps

²http://en.wikipedia.org/wiki/Test-driven_development

auf Code, den wir wiederholt schreiben und in eigene Libraries auslagern möchten. So kann solcher Code wiederverwendet und ggf. an andere Entwickler weitergegeben werden. Es hat sich mittlerweile eine sehr aktive Open Source Community um die iOS Entwicklung gebildet. Nicht zuletzt ist dies dem *CocoaPods* [3] Projekt zu verdanken.

CocoaPods ist ein Dependency Manager für Swift und Objective-C. Entwickler, die ihre Library zur Verfügung stellen möchten, registrieren ihr Repository in Form eines *CocoaPod*. Anschließend kann ein solcher Pod im eigenen Projekt integriert werden, indem eine Datei *Podfile* mit der Zeile `pod 'PodName'` erstellt und im Terminal `pod install` ausgeführt wird. Um bspw. mein Swift Logging Framework Evergreen [4] in ein iOS Projekt zu integrieren, kann einfach ein Podfile mit folgendem Inhalt erstellt und installiert werden:

```
1 source 'https://github.com/CocoaPods/Specs.git'
2 platform :ios, '8.0'
3 use_frameworks!
4 pod 'Evergreen'
```

CocoaPods vereinfacht den Prozess damit erheblich, eine Open Source Library (ggf. mit eigenen Dependencies) eines anderen Entwicklers einzubinden, unter möglichen Versionierungseinschränkungen aktuell zu halten und in Xcode zu integrieren. Die meisten Open Source Libraries der iOS Plattform sind mittlerweile als CocoaPod verfügbar. Einen Überblick über die beliebtesten Projekte gibt eine GitHub Suche nach den *Most Starred* Objective-C Repositories [5].

Natürlich gibt es auch andere Möglichkeiten, öffentlichen Code im eigenen Projekt zu integrieren:

- Das öffentliche Repository kann heruntergeladen und die Dateien dem Projekt manuell hinzugefügt werden, doch Aktualisierungen gestalten sich dann natürlich schwierig.
- *Git Submodules* [6] sind situationsbedingt eine sinnvolle Alternative oder Ergänzung zu CocoaPods. Diese verhalten sich wie eigene Git Repositories eingebettet in ihrem Parent-Repository und sind nützlich, wenn kein CocoaPod angeboten wird oder zeitgleich an dem eingebundenen Repository weitergearbeitet werden soll.

³<http://cocoapods.org>

⁴<https://github.com/viWiD/Evergreen>

⁵<https://github.com/search?l=Objective-C&q=stars%3A%3E0&ref=searchresults&type=Repositories>

⁶<http://git-scm.com/book/de/Git-Tools-Submodule>