

UNIVERSITÄT HEIDELBERG

SOMMERSEMESTER 2014

Softwareentwicklung für iOS mit Objective-C und Xcode

SKRIPT

NILS FISCHER

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

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Organisatorisches | 4 |
| 1.1 | Über meine Person | 4 |
| 1.2 | Über diesen Kurs | 4 |
| 1.3 | Hardware und Software | 5 |
| 1.4 | Über dieses Skript | 5 |
| 1.5 | Dokumentationen und Referenzen | 6 |
| 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 | 12 |
| 2.2.2 | Build & Run | 13 |
| 2.2.3 | Navigator | 14 |
| 2.2.4 | Editor | 14 |
| 2.2.5 | Inspektor | 17 |
| 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 | 20 |
| 2.3.3 | IBOutlets & IBActions | 21 |
| 2.4 | Dokumentation | 22 |
| 2.5 | Testen auf iOS Geräten | 23 |
| 2.5.1 | Der Provisioning Prozess | 24 |
| 2.5.2 | Provisioning mit Xcode | 25 |
| 3 | Objective-C | 29 |
| 3.1 | Grundlagen der Programmierung | 29 |
| 3.1.1 | Primitive Datentypen | 30 |
| 3.1.2 | Kommentare | 30 |
| 3.1.3 | Output | 30 |
| 3.1.4 | Einfache Operationen | 31 |
| 3.1.5 | Abfragen | 31 |
| 3.1.6 | Schleifen | 32 |

| | | |
|----------|--|-----------|
| 3.1.7 | Strings | 33 |
| 3.1.8 | String Formatierung | 33 |
| 3.2 | Grundlagen der objektorientierten Programmierung | 34 |
| 3.2.1 | Klassen & Objekte | 34 |
| 3.2.2 | Interface & Implementierung | 35 |
| 3.2.3 | Attribute | 35 |
| 3.2.4 | Methoden | 36 |
| 3.2.5 | Instanz- und Klassenmethoden | 37 |
| 3.2.6 | Polymorphie | 38 |
| 3.2.7 | Getter und Setter Methoden | 38 |
| 3.2.8 | Instanziierung von Objekten | 39 |
| 3.2.9 | Verfügbarkeit von Klassen | 40 |
| 3.3 | Symbolnamen & Konventionen | 41 |
| 3.4 | Einige wichtige Klassen | 42 |
| 4 | Projektstrukturierung | 44 |
| 4.1 | Versionskontrolle mit Git | 44 |
| 4.1.1 | Grundlagen der Kommandozeilensyntax | 44 |
| 4.1.2 | Git Repository & Commits | 45 |
| 4.1.3 | Branches | 45 |
| 4.1.4 | Zusammenarbeit mit Git & GitHub | 46 |
| 4.1.5 | Git in Xcode | 47 |
| 4.1.6 | Gitignore | 48 |
| 4.1.7 | Dokumentation | 49 |
| 5 | iOS App Architektur | 50 |
| 5.1 | iOS App Lifecycle | 50 |
| 5.1.1 | App States | 50 |
| 5.1.2 | Startprozess einer iOS App | 51 |
| 5.2 | Das Model-View-Controller Konzept | 54 |
| 5.3 | View Hierarchie | 55 |
| 5.3.1 | Frame und CGRect | 55 |
| 5.3.2 | UIView Objekte | 57 |
| 5.4 | Auto Layout | 57 |
| 5.4.1 | Constraints | 57 |
| 5.4.2 | Intrinsic Content Size | 58 |
| 5.4.3 | Auto Layout im Interface Builder | 59 |
| 5.4.4 | Auto Layout im Code | 60 |

Kapitel 1

Organisatorisches

1.1 Über meine Person

Mein Name ist Nils Fischer und ich bin Physikstudent im 4. Semester an der Uni Heidelberg. Mit der Softwareentwicklung für iOS Geräte beschäftige ich mich nun schon seit einigen Jahren und bin mit drei eigenen und sechs Apps für Kunden im App Store vertreten.

Ihr könnt mich jederzeit über meine Email ^[1] erreichen. Ich freue mich immer über Feedback und Vorschläge von euch.

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

1.2 Über diesen Kurs

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

Jeden Montag um 16h ct. während des Semesters wird es zunächst eine Vorlesungseinheit über ein neues Thema geben und ein zugehöriges Übungsblatt verteilt. Zu Beginn der folgenden Vorlesung werden dessen Lösungen besprochen und vorzugsweise von einem von euch vorgestellt. Der Übungsteil schließt sich direkt an die Vorlesung an und bietet für euch die Möglichkeit, das neue Übungsblatt zu bearbeiten und Fragen zur Vorlesung oder eigenen Projekten zu stellen.

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

¹n.fischer@stud.uni-heidelberg.de

²peter.fischer@ziti.uni-heidelberg.de

1.3 Hardware und Software

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

Wir arbeiten fast ausschließlich mit **Xcode**, Apple's Integrierter Entwicklungsumgebung (IDE), in Version 4.2 oder neuer (empfohlen Version 5.1) oder neuer). Die neueste Version kann und sollte schon vor Beginn des Kurses im Mac App Store (kostenlos) heruntergeladen und installiert werden [3].



Die Installation von Xcode enthält dann das iOS SDK (Software Development Kit) entsprechend in Version 5.0 (bei Xcode Version 4.2) bis zu 7.1 (bei Xcode Version 5.1) oder neuer.

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

Um eigene Apps im Simulator zu testen, ist keine weitere Konfiguration nötig. Zum Testen auf eigenen iOS Geräten (iPhone / iPad / iPod Touch) ist ein Lizenzierungsprozess erforderlich, den wir noch durchführen werden (s. S. 23, Abschnitt 2.5). Führt bitte ein Softwareupdate eurer Geräte auf die neueste Version durch.

1.4 Über dieses Skript

Dieses Skript wird im Verlauf des Semesters auf der Vorlesungsseite [4] 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 5.1 und dem iOS SDK 7.1. 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.

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

⁴<http://ios-dev-kurs.github.io/>

1.5 Dokumentationen und Referenzen

Zusätzlich zur in Xcode integrierten und online verfügbaren Dokumentation (s. S. 22, *Abschnitt 2.4*) bietet Apple einige Ressourcen für iOS Developer an:

iOS Dev Center ^[5] ist Apple's Onlineplattform für iOS Entwickler. Hier ist auch das Member Center zur Accountverwaltung und das Provisioning Portal zur Verwaltung der Certificates und Provisioning Profiles (s. S. 24, *Abschnitt 2.5.1*) zu finden.

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

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

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

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

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

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

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

⁸<https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide>

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

¹⁰<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. Siehe auch 1.3 für Informationen zu den Hardware- und Softwarevoraussetzungen für diesen Kurs.

Wir werden lernen, Xcode's zahlreiche Funktionen zu nutzen, um nicht nur effizient Programmcode zu schreiben, sondern auch unsere Programmierprojekte zu verwalten, Apps auf eigenen iOS Geräten zu testen, 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 Objective-C Code sehr und korrigiert Syntaxfehler.

2.1 Workspaces & Projekte

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

2.1.1 Neue Projekte anlegen

Mit `File > New > Project...` oder `⌘ + ⌥ + N` erstellen wir ein neues Projekt und wählen im erscheinenden Dialogfenster (s. S. 8, Abb. 2.1) ein Template. Zum Ausprobieren der Grundlagen von Objective-C eignet sich das `OS X > Application > Command Line Tool`. In diesem Dialogfenster finden wir mit `Cocoa Application` auch das Template, um Mac OS X Apps zu schreiben. Für iOS Apps verwenden wir die Templates unter `iOS > Application`. Es kann prinzipiell mit einem beliebigen Template begonnen werden, diese unterscheiden sich nur

in bereits vorkonfigurierten Programmelementen. Häufig ist es hilfreich, eines der Templates zu verwenden, das der Struktur der App entspricht, die wir programmieren wollen. Andernfalls kann mit `Empty Application` eine komplett leere App erstellt werden.

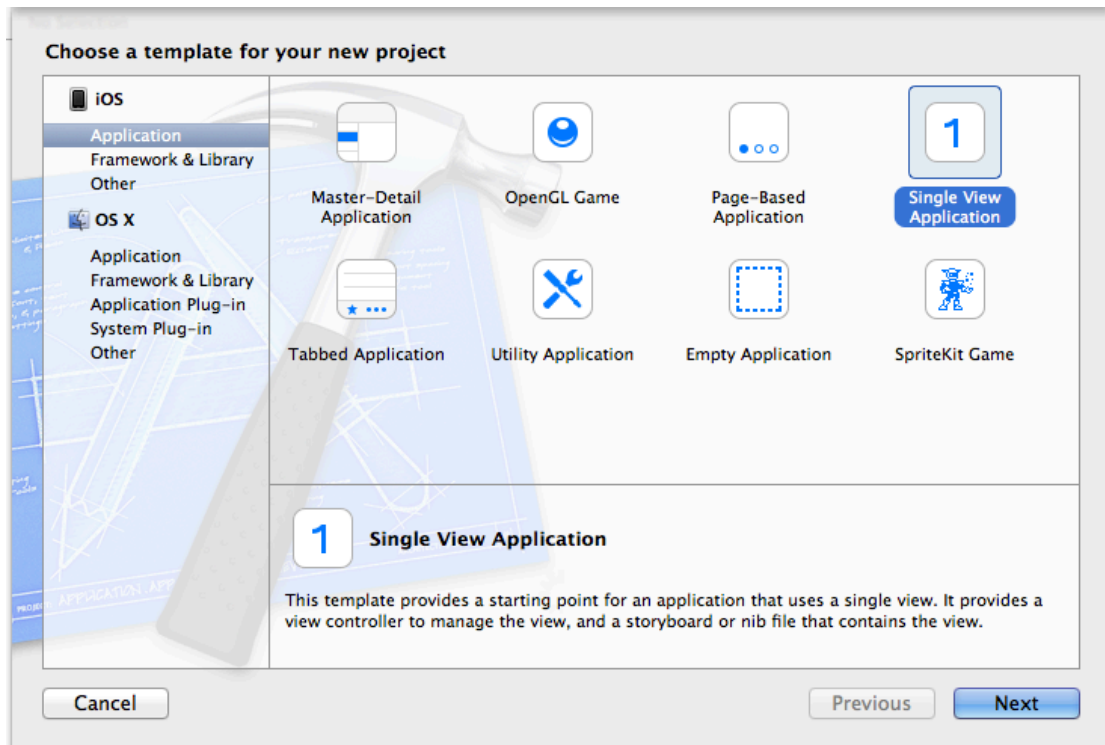


Abbildung 2.1: Ein neues Xcode-Projekt anlegen

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.deinname'** als Company Identifier.

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

Class Prefix wird den Dateinamen aller erstellten Dateien vorangestellt (s. S. 41, Abschnitt 3.3).

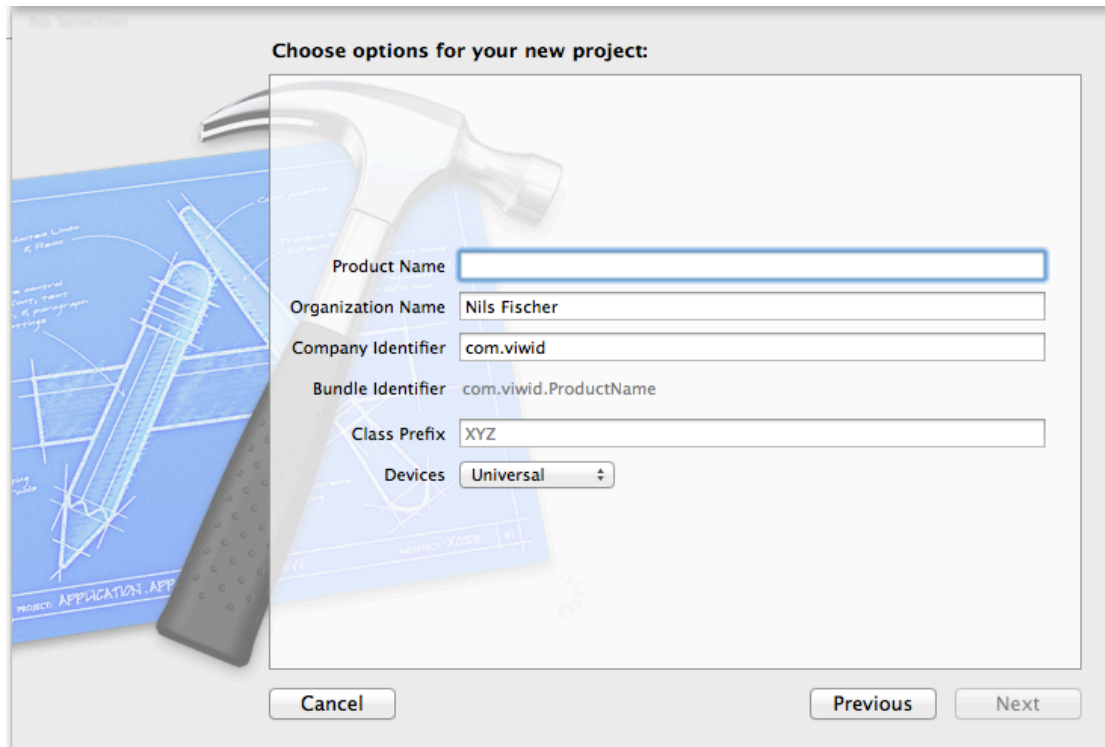


Abbildung 2.2: Ein neues Projekt konfigurieren

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. ??, Abschnitt ??).

Anschließend kann ein Speicherort für den Projektordner gewählt werden. Hier besteht zusätzlich die Möglichkeit, direkt ein Git Repository für das Projekt anzulegen. Git werden wir zu einem späteren Zeitpunkt noch thematisieren (s. S. 44, Abschnitt 4.1).

2.1.2 Targets und Products

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

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

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

2.1.3 Projekt- und Target-Konfiguration

Wenn das Projekt selbst im Project Navigator (s. S. 14, Abschnitt 2.2.3) ausgewählt ist, wird im Editor-Bereich die Projekt- und Target-Konfiguration angezeigt (s. S. 10, Abb. 2.3). Mit der Schaltfläche oben links kann das Projekt oder ein Target ausgewählt werden.

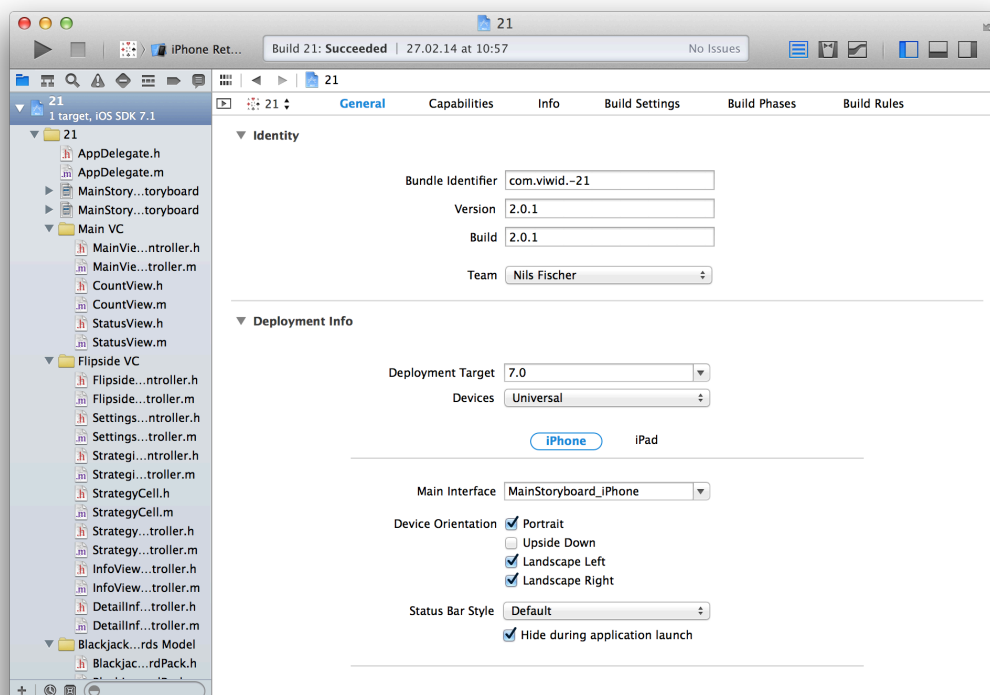


Abbildung 2.3: Die Projekt- und Targetkonfiguration wird angezeigt, wenn das Projekt links im Project Navigator ausgewählt ist

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.deinname.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. 23, 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 angepasst 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. 51, Abschnitt 5.1.2). In den meisten Fällen sollte hier die Storyboard-Datei für iPhone bzw. iPad ausgewählt werden.

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

Capabilities

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

Info

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

Build Settings

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

2.2 Benutzerinterface

Xcode's Interface ist in die in der Abbildung farbig markierten Bereiche aufgeteilt (s. S. 12, Abb. 2.4).

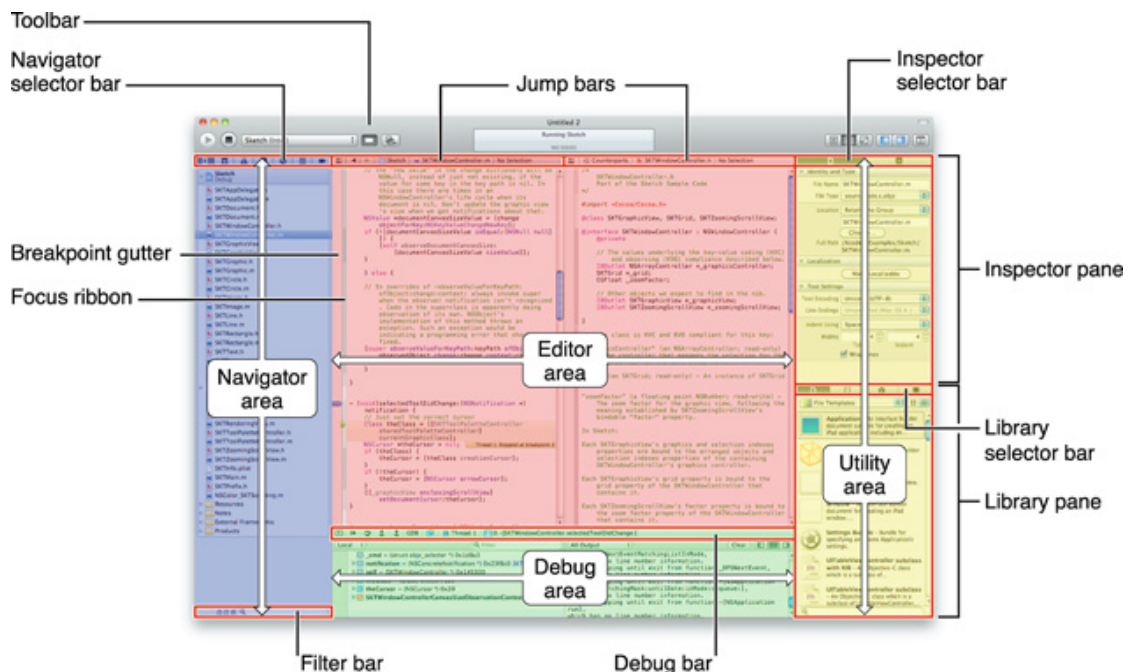


Abbildung 2.4: Xcode's Interface

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. 44, Abschnitt 4.1).


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 (sieben vor Xcode Version 5.0) 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.

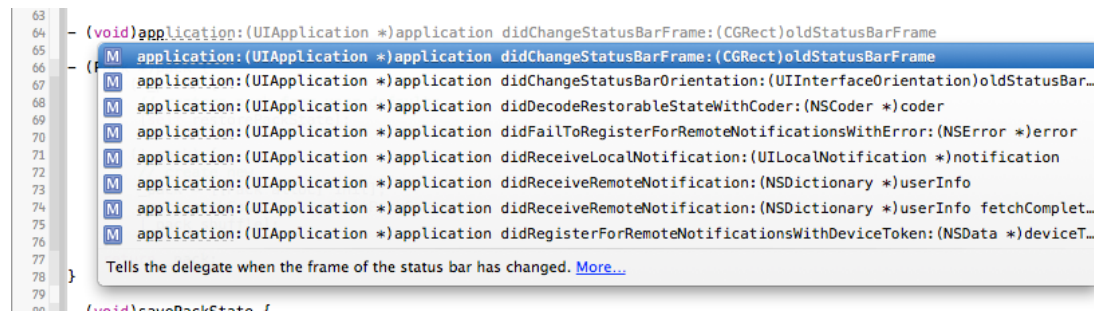


Abbildung 2.5: Der meiste Code wird mit der Autovervollständigung geschrieben

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

Objective-C ist eine sehr deskriptive Programmiersprache, deren Symbole nach Konventionen benannt sind (s. S. 41, Abschnitt 3.3). So können mit etwas Übung und Hilfe der Autovervollständigung auch unbekannte Symbole aus Apple's Frameworks gefunden werden, ohne erst die Dokumentation zu durchsuchen. Suchen wir beispielsweise eine bestimmte Konstante, die das Verhalten einer Animation eines `UIView`-Objekts bestimmt, so ist es typisch, die Autovervollständigung folgendermaßen zu verwenden:

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

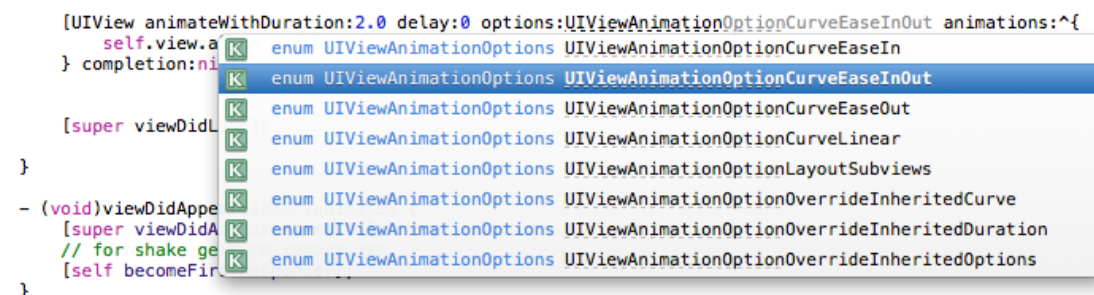


Abbildung 2.6: Auch unbekannte Symbole können mit der Autovervollständigung gefunden werden


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. 16, Abb. 2.8).

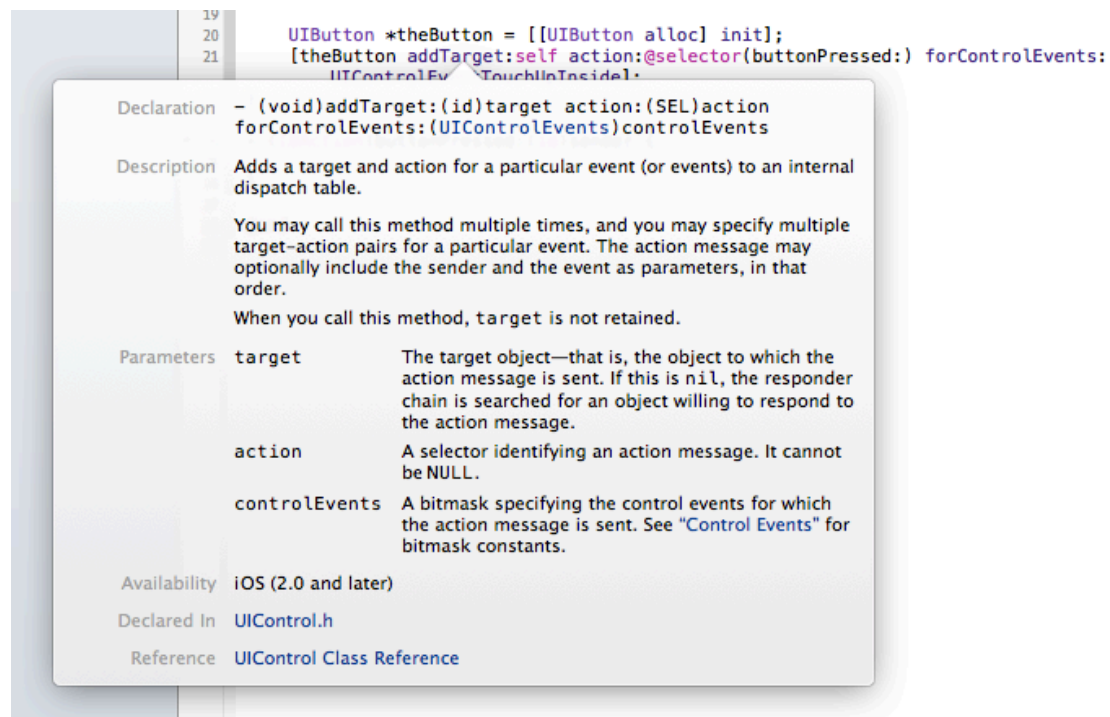






Abbildung 2.8: Alt-Klick auf ein Symbol zeigt eine kurze Definition

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

Jump bars & Open Quickly

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

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

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







Assistent


Im Assistent-Mode wird der Editor-Bereich zweigeteilt. Während der linke Teil die geöffnete Datei anzeigt, kann im rechten Teil mit Klick auf das erste Segment der Jump bar eine zugehörige Datei geöffnet werden. Empfehlenswert ist hier die Option *Counterpart*, die zu einer Main-Datei immer die entsprechende Header-Datei anzeigt (s. S. 35, Abschnitt 3.2.2). Diese Ansicht werden wir hauptsächlich verwenden, es stehen jedoch noch weitere situationsbedingte Optionen zur Verfügung.

Breakpoints

Mit einem Klick auf eine Zeilennummer in der Leiste rechts vom Editorbereich kann ein Breakpoint in dieser Codezeile gesetzt werden, sodass die App bei der Ausführung an dieser Stelle angehalten wird (s. S. 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  +  +  bis  +  +  erreichbar sind. Während Code geschrieben wird, sind hier nur zwei Tabs verfügbar:

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

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

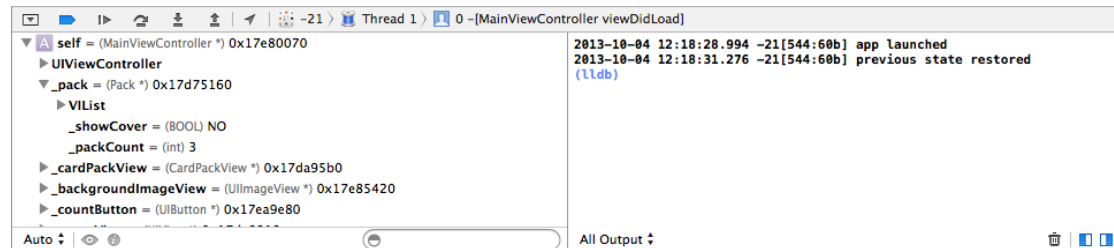


Abbildung 2.9: Im Debugger werden Konsolenausgaben und Situation der App angezeigt

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. Fährt man mit dem Cursor über ein Symbol, können Informationen über den aktuellen Status desselben angezeigt und in der Konsole ausgegeben werden. Zusätzlich zu herkömmlichen Werten der Variablen können sogar Bilder, die in den ausführenden Speicher geladen wurden, mit Quicklook angezeigt werden.

Da Code selten sofort so funktioniert wie wir möchten, ist 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. 19, 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.

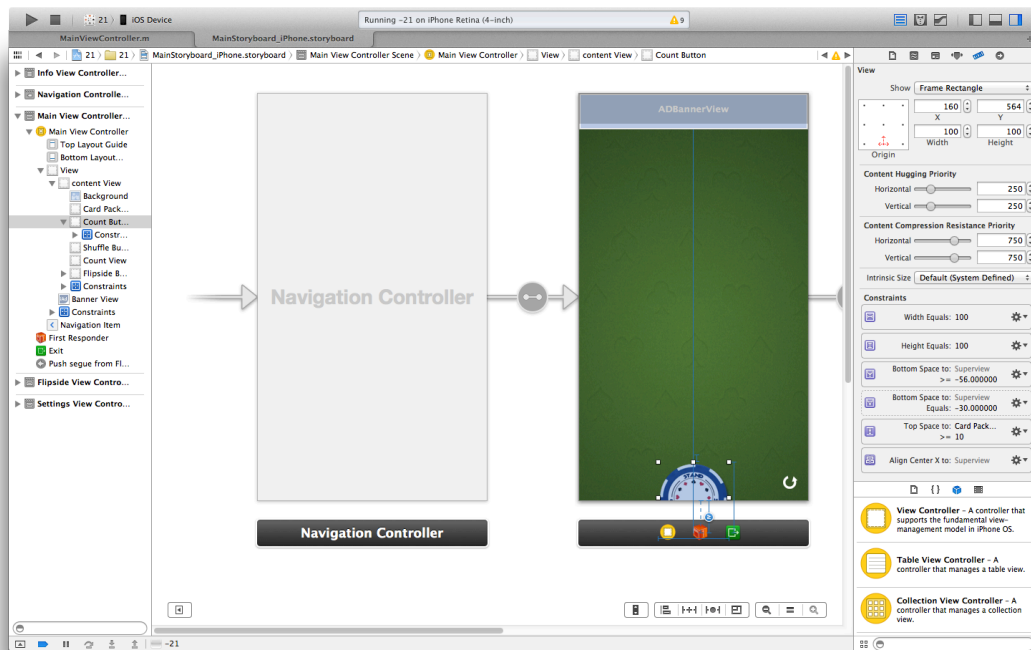



Abbildung 2.10: Um möglichst viel Platz zur UI-Gestaltung zu erhalten, verwenden wir für den Interface Builder eine angepasste Konfiguration

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

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

2.3.1 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.

Für Universal Apps erstellen wir ein Storyboard für die iPhone Version der App und eines für die iPad Version. In den Target-Einstellungen (s. S. 10, Abschnitt 2.1.3) wird das jeweils verwendete Storyboard ausgewählt.

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

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

2.3.2 Inspektor im IB-Modus

Im IB-Modus stehen im Inspektor zusätzlich zum File- und Quick-Help-Inspektor (s. S. 17, Abschnitt 2.2.5) vier weitere Tabs zur Verfügung, mit denen ein ausgewähltes Objekt im Interface Builder konfiguriert wird:

Identity Inspector dient dem Einstellen der Identität des Objekts, also hauptsächlich seiner Klasse (s. S. 34, Abschnitt 3.2.1).

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

Size Inspector enthält Optionen zu Größe und Position des Objekts.

Connections Inspector zeigt die verbundenen IBOutlets und IBActions des Objekts an (s. S. 21, Abschnitt 2.3.3).

2.3.3 IBOutlets & IBActions

Hinweis: Für diesen Abschnitt ist Kenntnis über **Attribute** (s. S. 35, Abschnitt 3.2.3) und **Methoden** (s. S. 36, Abschnitt 3.2.4) notwendig.

IBOutlets

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

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

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

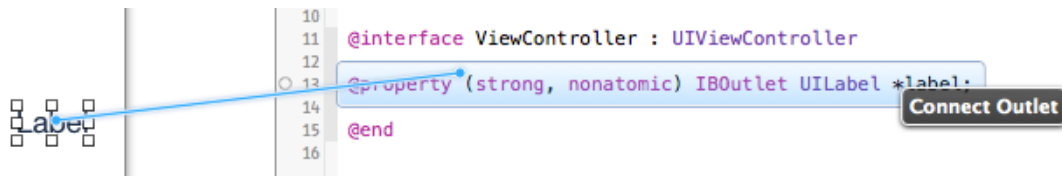


Abbildung 2.11: IBOutlets verbinden Interfaceelemente mit Properties im Code

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 **Automatic >> Klassenname.h** wählen, also die zugehörige Header-Datei, 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. 20, 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 IBOutlet's markierte Attribute werden nur sehr selten im öffentlich Interface in der Header-Datei deklariert, da meist nur innerhalb der Klasse auf die Interfaceelemente zugegriffen werden muss. Daher sollten sie stattdessen im privaten Interface in der Main-Datei definiert werden (s. S. 35, Abschnitt 3.2.2).

IBActions

IBActions funktionieren ähnlich wie IBOutlet's 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. 22, Abb. 2.12). **IBAction** ist als Rückgabewert der Methode äquivalent zu **void**.

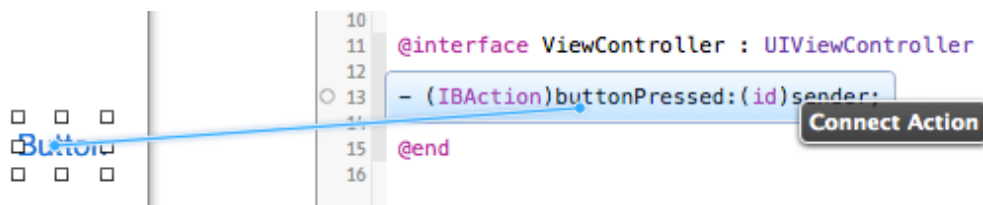



Abbildung 2.12: IBActions verbinden Events mit Methoden im Code

Die verbundene Methode wird dann ausgeführt, wenn das entsprechende Event ausgelöst wird. Das auslösende Objekt wird der Methode als Parameter **sender** des Typs **id** (beliebiger Typ) 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:@selector(buttonPressed:)
 forControlEvents:UIControlEventTouchUpInside];
```

## 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. 23, Abb. 2.13).

Dieses ist außerdem immer mit dem Tastenkürzel  $\text{⌘} + \text{⇧} + \text{?}$  (bzw.  $\text{⌘} + \text{⇧} + \text{⌵} + \text{B}$  auf deutschen Tastaturen) erreichbar.

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

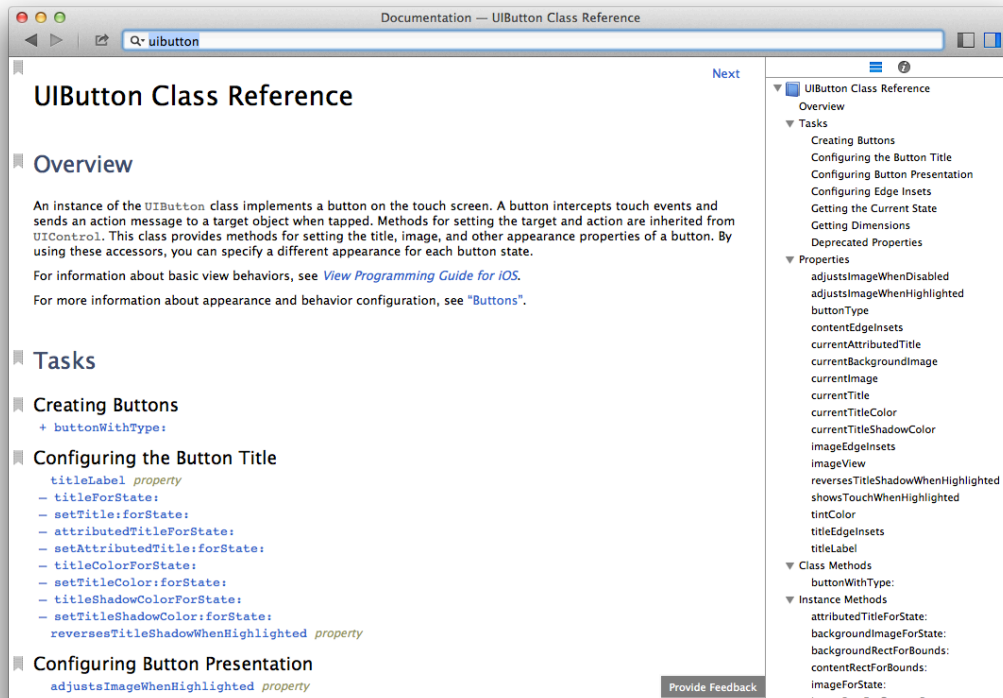


Abbildung 2.13: Xcode's integrierte Dokumentation ist beispielhaft strukturiert und sollte immer als erste Referenz verwendet werden

Die Dokumentation ist außerdem online in der iOS Developer Library <sup>[1]</sup> verfügbar. Eine Google-Suche nach dem Klassen- oder Symbolnamen führt meist direkt dorthin. Apple bietet online zusätzlich noch viele weitere Ressourcen für iOS Developer an (s. S. 6, Abschnitt 1.5).

## 2.5 Testen auf iOS Geräten

Der Simulator eignet sich sehr gut zum Testen eurer Apps. Er läuft auf der Architektur eures Mac's und ist daher in den meisten Fällen deutlich schneller als ein iOS Gerät, verfügt jedoch nicht über alle Funktionen eines solchen. Außerdem gibt es einen großen Unterschied zwischen der Bedienung einer App mit Touchpad oder Maus im Vergleich zu einem Touchscreen. Beispielsweise wird die Größe und Position von Schaltflächen beim Testen

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

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. 24, Abb. 2.14).

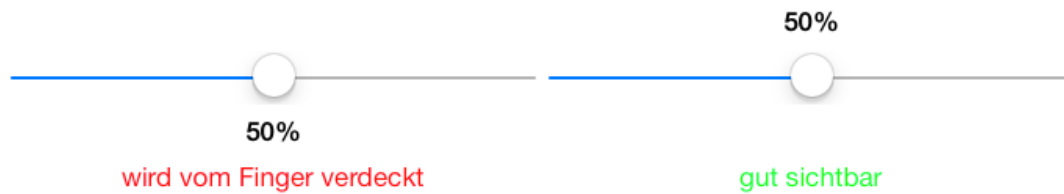


Abbildung 2.14: Unter einer Schaltfläche positionierte Interfaceelemente sind bei der Bedienung meistens vom Finger bedeckt. Solche Probleme werden häufig erst beim Testen auf realen iOS Geräten entdeckt.

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.

Für diesen Kurs ist es empfehlenswert, die neueste Version von Xcode, dem iOS SDK und der auf dem Gerät installierten Software zu verwenden (s. S. 5, Abschnitt 1.3). Leider ist auf den Macs im Medienzentrum nur die Xcode Version 4.2 mit iOS SDK 5.0 installiert. Daher können mit diesen Macs keine Geräte mit einer neueren Softwareversion als iOS 5.0 zum Testen verwendet werden.

### 2.5.1 Der Provisioning Prozess

**Hinweis:** Dieser Abschnitt stellt die Grundlagen des Provisioning Prozesses dar. Xcode vereinfacht diesen Vorgang mittlerweile sehr (s. S. 25, Abschnitt 2.5.2).

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

Der Developer Mac erstellt zunächst eine **Signing Identity**, die aus einem public-private-Key-Paar besteht und im Schlüsselbund (Keychain) eures Macs gespeichert wird. Mit dem public Key wird dann ein **Certificate** angefordert, das an Apple und das Developer Team übermittelt und bestätigt wird. Dieses wird anschließend ebenfalls im Schlüsselbund gespeichert. Es wird zwischen **Development Certificates**, die einen einzelnen Entwickler identifizieren und das Testen von Apps auf dessen Geräten erlauben, und **Distribution Certificates**, die der Identifikation des Development Teams und zur Veröffentlichung von Apps



im App Store dienen, unterschieden. Die Certificates sind online im Member Center [2] einsehbar.

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

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






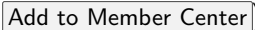
Für die Entwicklung können ebenfalls Development Provisioning Profiles erstellt werden, die genau auf eine Bundle ID bezogen sind. Diese werden **explicit** genannt und sind erforderlich, wenn Services wie iCloud verwendet werden, die eine exakte Identifikation benötigen. Solange dies nicht erforderlich ist, kann anstatt der Bundle ID auch ein Asterisk (\*) verwendet werden. Ein solches Development Provisioning Profile wird auch **wildcard** genannt und kann für beliebige Bundle IDs verwendet werden.

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

## 2.5.2 Provisioning mit Xcode

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

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

1. Erstellt eine Apple ID [3] oder verwendet, soweit vorhanden, eure existierende Apple ID
2. Schließt euer iOS Gerät mit dem USB-Kabel an.
3. Öffnet in Xcode den **Organizer** mit  +  + .
4. Wählt dort den Tab  und in der linken Seitenleiste euer Gerät aus (s. S. 26, Abb. 2.15).
5. Mit den Schaltflächen  oder  können nur Admins des Developer Teams ihre Geräte direkt im Team registrieren. Stattdessen kopiert bitte die UDID eures iOS Geräts, also die lange Zeichenfolge mit dem Label *Identifier*.

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

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

6. Teilt mir **eure registrierte Apple ID** und die **UDIDs** aller eurer **iOS Geräte** per Email <sup>[4]</sup> mit, sodass ich euch in das Developer Team einladen und die Geräte hinzufügen kann. Verwendet dazu bitte die Email, mit der ihr euch zum Kurs angemeldet habt.
7. Mit dem Einladungslink, den ihr anschließend per Email von Apple erhaltet, könnt ihr dem Developer Team beitreten. Ihr müsst euch im erscheinenden Dialog nicht noch einmal registrieren, wählt hier einfach 'Sign in' und loggt euch mit der verwendeten Apple ID ein.

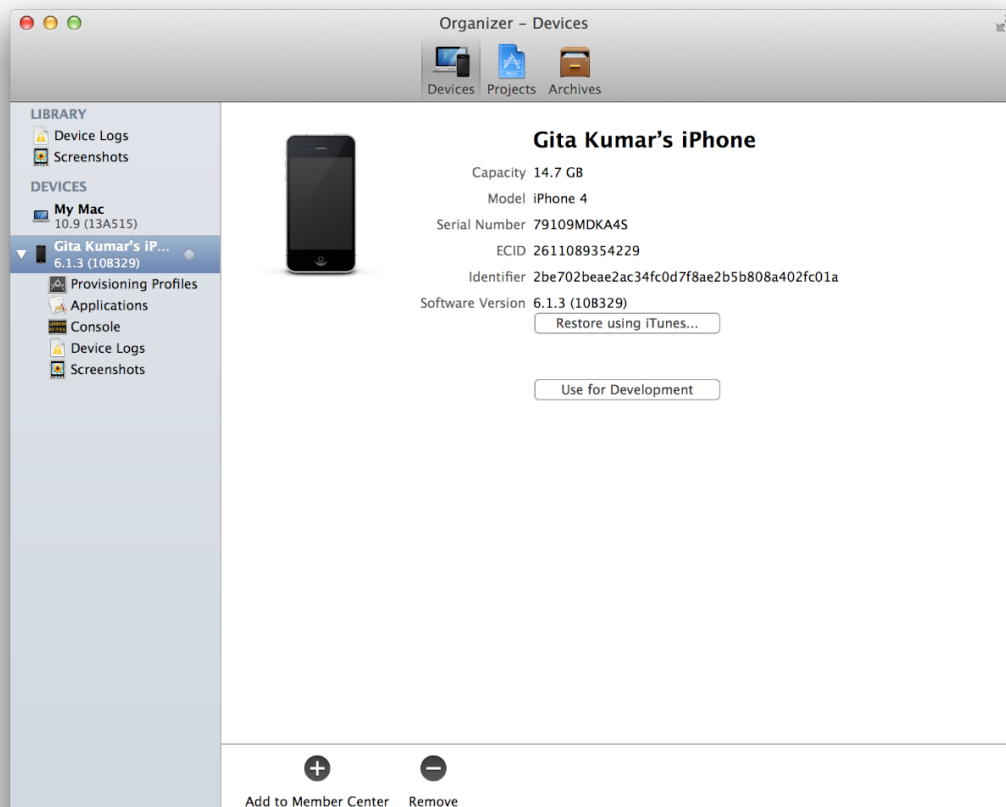


Abbildung 2.15: Im Organizer können die iOS Geräte verwaltet werden. Developer Team Admins können die Schaltfläche *Use for Development* oder *Add to Member Center* verwenden, andernfalls muss die UDID des Geräts manuell dem Team hinzugefügt werden. (Bild aus Apple's Dokumentation <sup>[5]</sup>)

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

<sup>4</sup>[n.fischer@stud.uni-heidelberg.de](mailto:n.fischer@stud.uni-heidelberg.de)

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

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

Nun integriert euren Apple Developer Account in Xcode:

1. Öffnet in Xcode **Preferences** » **Accounts** und fügt euren Apple Developer Account dort hinzu (s. S. 27, Abb. 2.16).
2. Auf der rechten Seite erscheinen die Developer Teams, denen ihr angehört. Wählt das Team der Uni Heidelberg aus und klickt **View Details**.
3. Im erscheinenden Dialog könnt ihr nun eure Certificates und Provisioning Profiles verwalten (s. S. 28, Abb. 2.17). Die Liste der Provisioning Profiles dient hier nur zur Information. Ihr benötigt nur ein Development Certificate, das ihr mit Klick auf **+ iOS Development** anfordern könnt. Es muss nun zunächst von mir bestätigt werden, daher kann zunächst eine Fehlermeldung erscheinen.
4. Ein Klick auf das **Refresh**-Symbol unten links aktualisiert die Liste und lädt verfügbare Certificates und Provisioning Profiles automatisch herunter. Diese sind außerdem jederzeit online im Member Center einsehbar [7].

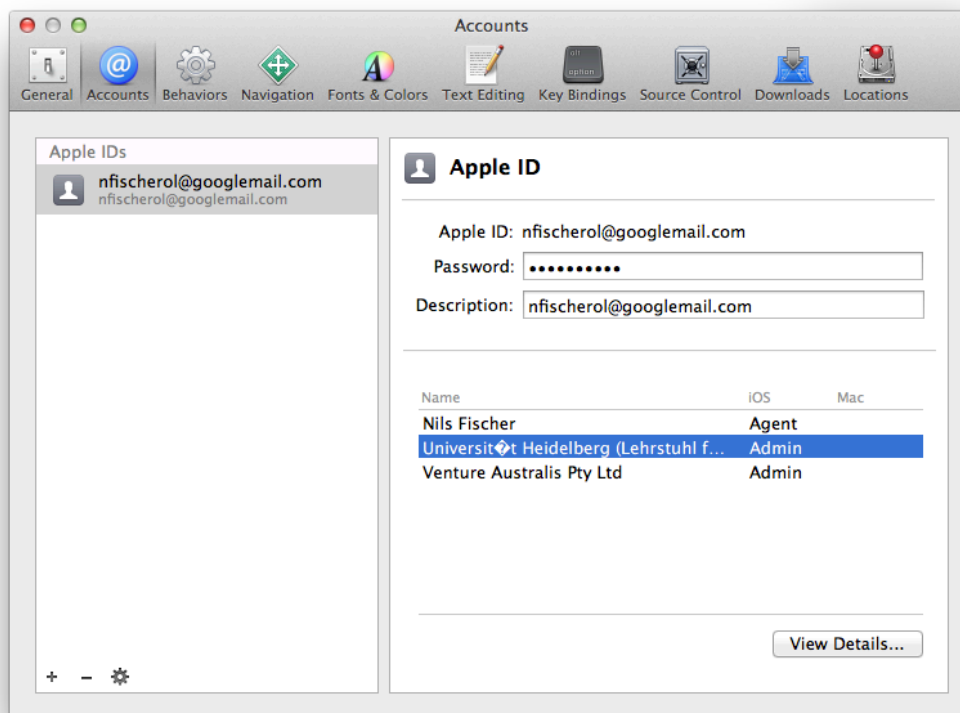


Abbildung 2.16: In den Einstellungen kann der Apple Developer Account hinzugefügt werden

Wenn Certificates oder Devices im Developer Team hinzugefügt werden, generiert Xcode automatisch auch ein wildcard Team Provisioning Profile, das euer iOS Device zum Tes-

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

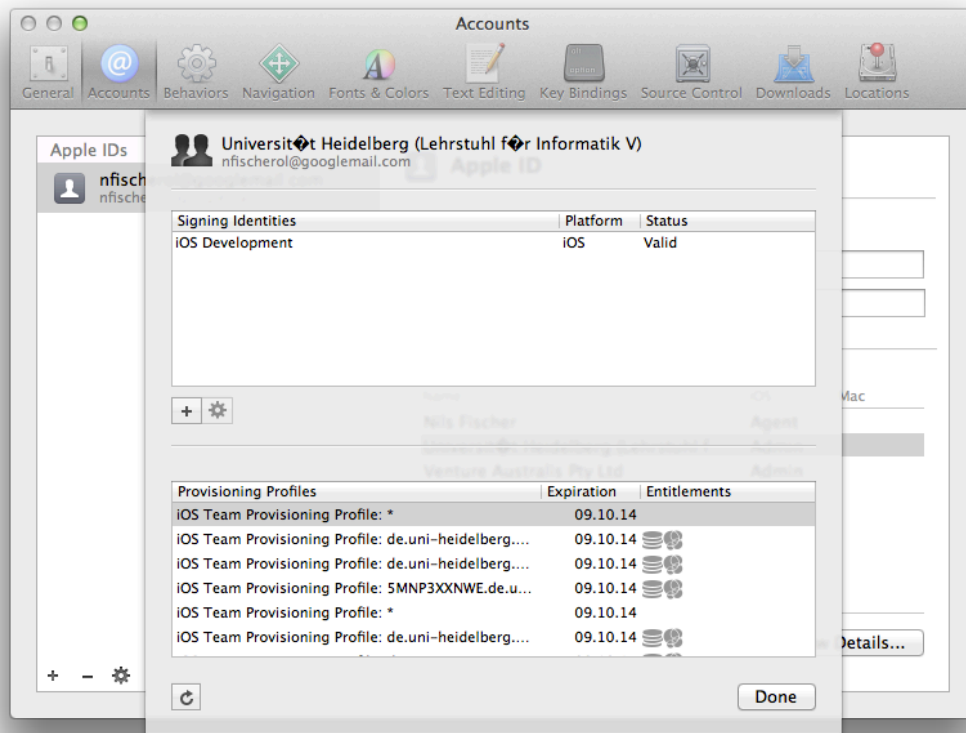


Abbildung 2.17: Mit Xcode knnen Sie Ihre Certificates und Provisioning Profiles verwalten

ten von beliebigen Apps autorisiert. Auerdem werden fr solche Apps, die es erfordern, zustzlich explicit Team Provisioning Profiles generiert (s. S. 24, Abschnitt 2.5.1).

Nun knnen Sie Ihre Apps auf Ihrem iOS Gert installieren und testen:

1. Stellt sicher, dass in der Target-Konfiguration unser Development Team ausgewhlt ist und die Bundle ID dem beschriebenen Schema entspricht (s. S. 10, Abschnitt 2.1.3).
2. Whlt Ihr iOS Device als Zielsystem und fhrt einen *Build & Run* aus (s. S. 13, Abschnitt 2.2.2).
3. Die App wird nun auf dem angeschlossenen Gert ausgefhrt!

## Kapitel 3

# Objective-C

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

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

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

### 3.1 Grundlagen der Programmierung

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

Ein Befehl kann beispielsweise eine einfache Zuweisung sein:

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

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

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

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

### 3.1.1 Primitive Datentypen

Die wichtigsten primitiven Datentypen sind:

**int** integer: Ganze Zahl, z.B. `int a = -42;`

**uint** unsigned integer: Nichtnegative ganze Zahl, z.B. `uint a = 1;`

**char** character: Einzelnes Zeichen, z.B. `char a = 'x';`

**float** floating-point number: Dezimalzahl, z.B. `float a = 3.14;`

**double** double precision floating-point: Sehr lange Dezimalzahl

**BOOL** boolean: Entweder wahr (`YES`) oder falsch (`NO`), z.B. `BOOL a = YES;`

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

### 3.1.2 Kommentare

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

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

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

Verwendet in den Übungen häufig Kommentare!

### 3.1.3 Output

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

### 3.1.4 Einfache Operationen

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

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

Typen können mit der sog. **cast** Operation ineinander umgewandelt werden, bei der der Wert als der Typ interpretiert wird, der in Klammern davor angegeben wird:

Mit `int a = (int)3.14;` hat `a` nun den Wert 3.

```

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

```

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

```

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

```

### 3.1.5 Abfragen

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

```

1 // Sei year eine Variable, z.B.:
2 int year = 2014;
3
4 if (year < 2014) {
5 NSLog(@"Vergangenheit");
6 } else if (year == 2014) {
7 NSLog(@"Gegenwart");
8 } else if (year > 2014) {
9 NSLog(@"Zukunft");
10 } else {
11 NSLog(@"unmöglich!");
12 }

```

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

Logische Operatoren:

**a == b** gleich

**a != b** ungleich

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

**!a** logische Negation

**a && b** logisches Und

**a || b** logisches Oder

### 3.1.6 Schleifen

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

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

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

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

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

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

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

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



### 3.1.7 Strings

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

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

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

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

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

### 3.1.8 String Formatierung

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

`%i` für `int` und `uint`

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

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

`%c` für `char`

`%@` für Strings

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

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

## 3.2 Grundlagen der objektorientierten Programmierung

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

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

### 3.2.1 Klassen & Objekte

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

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

Betrachten wir nun beispielsweise folgende Klasse:

#### Person.h

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

#### Person.m

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

```

11 }
12
13 @end

```

### 3.2.2 Interface & Implementierung

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

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

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

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

Zusätzlich kann die Main-Datei noch ein privates Interface enthalten, auf das nur innerhalb der Klasse zugegriffen werden kann. Dieses schreiben wir in der Main-Datei vor der Implementierung. Hier werden Attribute und Methoden deklariert, die nur innerhalb der Klasse verwendet werden. Bei einer eleganten Programmierweise werden nur solche Attribute und Methoden im öffentlichen Interface präsentiert, die zur Kommunikation mit dem restlichen Programm notwendig sind.

### 3.2.3 Attribute

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

Im Codebeispiel oben (s. S. 34, Abschnitt 3.2.1) spezifizieren wir, dass eine Person immer eine Variable name des Typs `NSString` hat:

```

1 @property (strong, nonatomic) NSString *name;

```

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

```

1 @property (strong, nonatomic) NSString *name; // Wenn die Person "stirbt",
 verfällt auch ihr Name: strong
2 @property (weak, nonatomic) Person *friend; // Eine hier referenzierte Person
 lebt dagegen weiter: weak

```

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

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

Typische Attributdeklarationen im Interface sehen also wie folgt aus:

```

1 // Objekte mit starker Referenz, die der Klasse konzeptionell direkt angehören
 und erst freigegeben werden, wenn das auch das Mutterobjekt freigegeben
 wird:
2 @property (strong, nonatomic) NSString *name;

1 // Objekte mit schwacher Referenz, deren Existenz im Speicher nicht durch das
 Mutterobjekt bedingt sein soll
2 @property (weak, nonatomic) NSObject *delegate;

1 // Primitive Datentypen
2 @property (nonatomic) int number;
```

### 3.2.4 Methoden

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

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

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

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

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

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

```

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

Im Codebeispiel (s. S. 34, Abschnitt 3.2.1) geben wir bspw. an, dass die Methode `sayHi` keinen Rückgabewert hat: `(void)`. Andernfalls wird stattdessen der Typ des Rückgabewerts angegeben, also z.B. `(int)` oder `(NSString*)`.

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

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

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

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

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

### 3.2.5 Instanz- und Klassenmethoden

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

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

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

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

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

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

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

### 3.2.6 Polymorphie

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

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

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

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

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

### 3.2.7 Getter und Setter Methoden

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

Für jedes Attribut wird automatisch eine **Getter**- und einer **Setter**-Methode generiert.

Die Getter-Methode ist einfach eine Methode mit gleichem Namen wie das Attribut, die den Wert desselben als Rückgabewert liefert. Die Setter-Methode stellt dem großgeschriebenen Attributnamen ein `set` voran und nimmt als Parameter den neuen Wert an.

Sie sind somit definiert als:

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

Die Verwendung erfolgt wie bei regulären Methoden:

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

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

```

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

```

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

```

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

```

Anders als bei der Dot-Syntax werden in diesem Fall jedoch **nicht** die Getter- und Setter-Methoden aufgerufen. Außer in bestimmten Situationen, von denen eine unten beschrieben ist, sollte immer die Methoden- oder Dot-Syntax verwendet werden.

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

```

1 // Getter
2 - (Type)variable {
3 // do something
4 return _variable;
5 }

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

```

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

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

```

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

```

### 3.2.8 Instanziierung von Objekten

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

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

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

### 3.2.9 Verfügbarkeit von Klassen

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

#### Forward Declarations

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

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

#### A.h

```
1 @class B; // Forward Declaration, um B im Interface zu verwenden
2
3 @interface A : NSObject
4
5 @property (strong, nonatomic) B *b;
6
7 @end
```

#### A.m

```
1 #import "B.h" // Echter Import erst in der Main-Datei
2
3 @implementation A
4
5 @end
```

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



### 3.3 Symbolnamen & Konventionen

Symbolnamen müssen in Objective-C bestimmte Bedingungen erfüllen:

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

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

- Variablen- und Methodennamen beginnen mit einem Kleinbuchstaben, Klassennamen mit einem Großbuchstaben
- Zur Repräsentation mehrerer Wörter werden keine Unterstriche verwendet. Stattdessen wird der Anfangsbuchstabe jedes folgenden Worts großgeschrieben (*camel case*). Beispiele:
  - `isEnabled`
  - `objectForIndex:`
  - `viewDidLoad`
- Ein Methodenname beginnt mit einer Beschreibung des Rückgabewerts oder der ausführenden Aktion und bezieht ihre Argumente in den Namen ein. Beispiele:
  - `filteredArrayUsingPredicate:`
  - `pushViewController:animated:`
- Verwendet einen Klassenprefix für größere Projekte, sodass jede Klasse eindeutig benannt ist, denn es gibt keine Namespaces in Objective-C. Verwendet mindestens drei Großbuchstaben, die eurem Company Identifier und/oder Product Name ähneln und beginnt jede Klasse diesem Prefix. Andernfalls besteht die Gefahr, dass Dateien gleichen Namens aus anderen Projekten zu Konflikten führen. Allen von Apple zur Verfügung gestellten Dateien steht ein 'NS' (für NextStep, die Programmiersprache des Betriebssystems aus dem das heutige Mac OS X hervorging) oder 'UI' (für User Interface, hauptsächlich bei iOS-relevanten Dateien) voran. Beispiele:
  - `NSArray`
  - `UIView`
- Methodennamen besitzen kein Prefix und müssen nur innerhalb der Klasse eindeutig sein. In verschiedenen Klassen wird sogar häufig der gleiche Methodenname verwendet, wenn diese Methoden einen ähnlichen Sinn erfüllen. So überschreiben bspw. viele Klassen die Methode `description`, deren Rückgabewert vom Typ `NSString` eine Textrepräsentation des Objekts darstellen soll.

### 3.4 Einige wichtige Klassen

In Objective-C sind viele Grundelemente der Programmierung Objekte. Dazu gehören die folgenden häufig verwendete Datenstrukturen. Diese sind ausführlich in der Dokumentation beschrieben (s. S. 22, Abschnitt 2.4).

#### NSString

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

Häufig verwenden wir die Klassenmethode `stringWithFormat:`, um Werte von Variablen nach der Syntax der String Formatierung einzubinden (s. S. 33, Abschnitt 3.1.8).

```
1 NSString *s = @"text";
2 NSString *t = [NSString stringWithFormat:@"number %i (%@)", 1, @"one"];
```

#### NSNumber

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

```
1 NSNumber *a = @(3.14);
2 NSNumber *b = @(YES);
3 float c = [a floatValue]; // 3.14
4 int d = [b intValue] // 1
5 BOOL e = [b boolValue] // YES
```

#### NSArray

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

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

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

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

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

```
1 NSArray *a = @[@"one", @"two", @"three"];
2 NSString *s = [a objectAtIndex:0];
3 NSMutableArray *m = [a mutableCopy];
4 [m addObject:@"four"];
```

## NSDictionary

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

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

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

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

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

## Kapitel 4



# Projektstrukturierung

### 4.1 Versionskontrolle mit Git

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

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

#### 4.1.1 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.

### 4.1.2 Git Repository & Commits

Während wir an unserem Projekt arbeiten, können wir Git verwenden, 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. Befindet sich der Code in einem akzeptablen Zwischenzustand, können wir die Änderungen mit einem Commit im Repository sichern:

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

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

### 4.1.3 Branches

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

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

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

```
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. So kann bspw. schnell ein Fehler in der veröffentlichten Version in `master` behoben und anschließend wieder zum Feature Branch gewechselt werden.

### 4.1.4 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. 49, Abschnitt 4.1.7) enthält detaillierte Beschreibungen zu Remote Repositories.

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

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

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

## 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 [2]. 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
```

### 4.1.5 Git in Xcode

Versionskontrolle ist tief in Xcode integriert und in vielen Kontextmenüs und Schaltflächen präsent. Das Menü `Source Control` stellt bspw. einige Benutzeroberflächen zu Git Befehlen zur Verfügung, die die Kommandozeilenbedienung ersetzen können. Sehr hilfreich ist jedoch hauptsächlich der Version Editor (s. S. 48, Abb. 4.1), der mit der Schaltfläche rechts in der Toolbar alternativ zum Assistant Editor angezeigt werden kann. Im rechten Editorbereich wird dann die Version der links angezeigten Datei zu einem früheren Commit angezeigt

---

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

<sup>2</sup><https://github.com/iOS-Dev-Kurs>

und Änderungen visualisiert. Mit der Uhr-Schaltfläche unten in der Mittelleiste können wir frühere Commits auswählen.

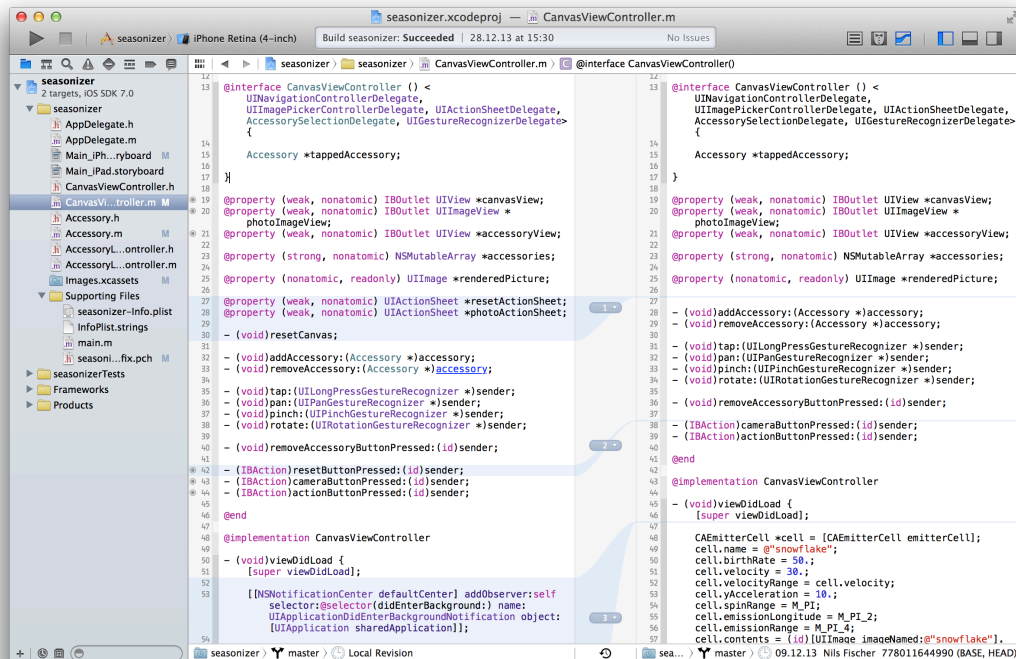


Abbildung 4.1: Der Version Editor zeigt die links geöffnete Datei zu einem früheren Commit an und visualisiert Änderungen

#### 4.1.6 Gitignore

Elemente 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* [3] zu finden. Kopiert für Xcode Projekte die Vorlagen *Xcode* [4] und *OSX* [5] in die .gitignore Datei. Letztere Vorlage enthält bspw. die Zeile *.DS\_Store*. So heißen versteckte Dateien in Mac OS X, die dem Betriebssystem verschiedene Informationen über den Ordner bereitstellen [6] und nicht in das Git Repository

<sup>3</sup><https://github.com/github/gitignore>

<sup>4</sup><https://github.com/github/gitignore/blob/master/Global/Xcode.gitignore>

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

<sup>6</sup>[http://en.wikipedia.org/wiki/.DS\\_Store](http://en.wikipedia.org/wiki/.DS_Store)



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"
```

#### 4.1.7 Dokumentation

Git [7] ist erstklassig dokumentiert und bietet neben Tutorials und einem Übungsmodus [8] eine umfassende Dokumentation [9]. Hier sollte bei Bedarf unbedingt nachgeschlagen werden.

---

<sup>7</sup><http://git-scm.com>

<sup>8</sup><http://try.github.com/>

<sup>9</sup><http://git-scm.com/book>

# Kapitel 5

## iOS App Architektur

### 5.1 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.

#### 5.1.1 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.

---

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

### 5.1.2 Startprozess einer iOS App

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

```

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

```

Die `main()` Methode wird auch als **Entry Point** bezeichnet und besitzt entsprechende Äquivalente in C und C++.


3. `main()` ruft `UIApplicationMain()` auf. Mit einem -Klick auf den Methodennamen können wir die Dokumentation dieser Methode in Apple's `UIKit` Framework anschauen (s. S. 51, Abb. 5.1).



Abbildung 5.1: Die integrierte Dokumentation ist hilfreich, um den App Lifecycle zu verstehen

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

U.a. stellt dieses Objekt die Verbindung zum Bildschirm her, erzeugt ein `UIWindow` zur Darstellung der Benutzeroberfläche und empfängt Benutzereingaben. Wir müssen für diese Klasse keinen weiteren Code schreiben, damit die Architektur der App

funktioniert. Über den Aufruf einer Klassenmethode nach dem sog. *Singleton Pattern* (s. S. ??, Abschnitt ??) kann jederzeit auf das `UIApplication` Objekt zugegriffen werden:

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

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

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

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

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

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

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

Im Verlauf des Startvorgangs und während der Laufzeit der App werden bestimmte Methoden des Application Delegates aufgerufen. Diese informieren bspw. über den Wechsel der oben beschriebenen States und sind im Übersichtsdiagramm dargestellt (s. S. 53, Abb. 5.2).

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

Dieses Storyboard wird geladen und dessen Initial Scene angezeigt. Die im Storyboard enthaltenen Objekte werden dabei instanziiert und die Connections (`IBOutlet`s und `IBAction`s) hergestellt.

7. Nun übernehmen unsere eigenen Klassen die Kontrolle. Unsere Implementierung des App Delegates wird hier häufig verwendet, um benötigte Datenstrukturen zu initialisieren oder die Benutzeroberfläche ohne Verwendung eines Storyboards aufzusetzen. Außerdem ist das App Delegate verantwortlich dafür, auf eine Veränderung der App States (s. S. 50, Abschnitt 5.1.1) zu reagieren und bspw. rechenintensive Prozesse der Benutzeroberfläche wie Animationen zu pausieren, wenn die App in den Hintergrund tritt.

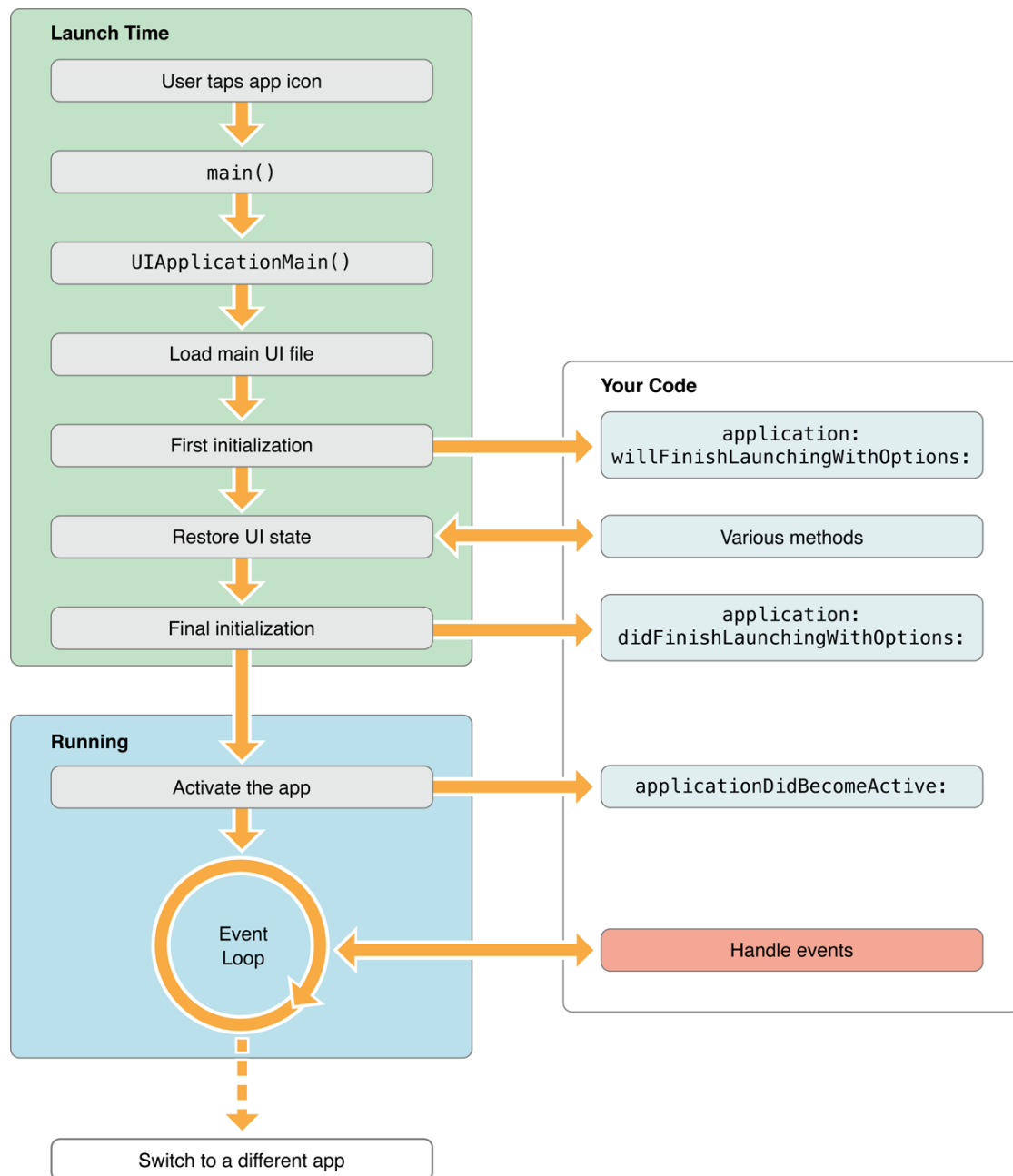


Abbildung 5.2: Übersicht der Prozesse beim Starten einer iOS App (aus dem iOS App Programming Guide)

## 5.2 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. 54, Abb. 5.3). Durch diese Modularisierung bleibt ein Softwareprojekt flexibel und erweiterbar und einzelne Komponenten können leicht wiederverwertet werden.

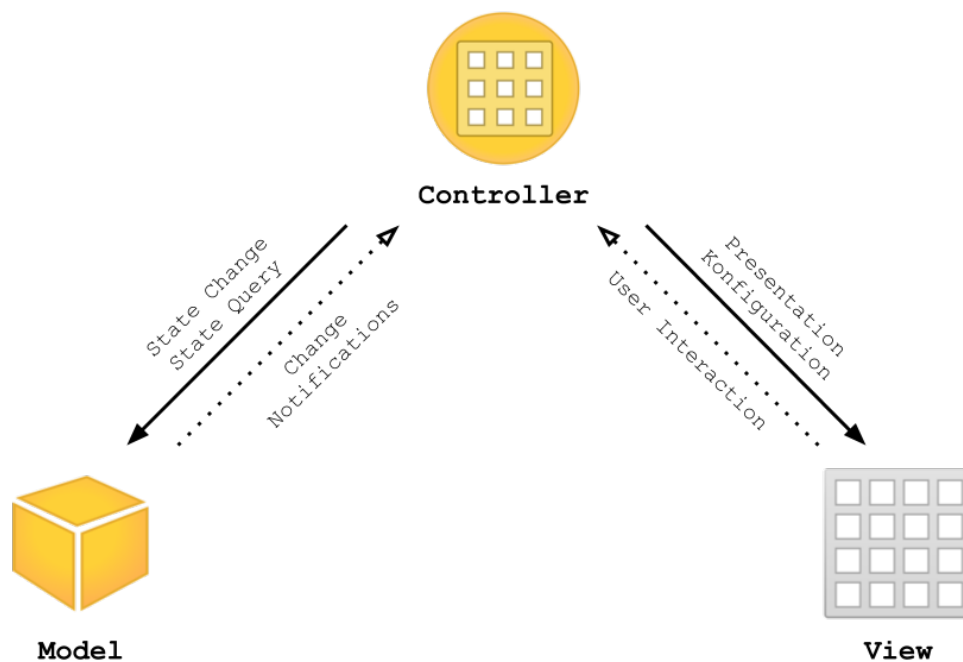


Abbildung 5.3: Das MVC Konzept trennt Modell, Präsentation und Steuerung

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

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

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

**Steuerung (Controller)** Der Controller verwaltet die Views und reagiert auf Benutzereingaben. Zur Laufzeit der App übernehmen Subklassen von `UIViewController` die

Kommunikation zwischen Model und View. Meist repräsentiert jeweils ein View Controller eine bestimmte Ansicht auf dem Bildschirm. In Reaktion auf Benutzereingaben stellt der View Controller Anfragen an das Model und konfiguriert die Views.

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

## 5.3 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. 56, Abb. 5.4). An oberster Stelle der Hierarchie steht dabei ein Objekt der Klasse `UIWindow`: `UIView`, das von dem Application Object verwaltet wird (s. S. 51, Abschnitt 5.1.2).

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

### 5.3.1 Frame und CGRect

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

Der Ursprung des Koordinatensystems liegt dabei immer in der oberen linken Ecke der Superview (s. S. 56, Abb. 5.5) mit einer horizontalen x-Achse und vertikalen y-Achse in positiver Richtung nach rechts unten.

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



Abbildung 5.4: Jedes Objekt der `UIView` Klasse dient wieder als Container für weitere Objekte dieser Klasse

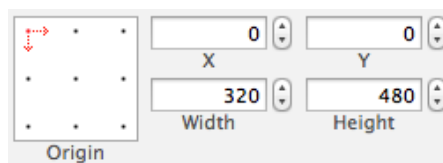


Abbildung 5.5: Der Ursprung des Koordinatensystems von `UIView` liegt in der oberen linken Ecke

```

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

```

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



### 5.3.2 UIView Objekte

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

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

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

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

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

### 5.4.1 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

$$y = m * x + b \quad (5.1)$$

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

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

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

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

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

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

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

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

Bei der Konstruktion der Constraints sollte unbedingt auf ein eindeutiges Layout geachtet werden. Ein solches liegt vor, wenn jeder Subview ein eindeutiger Wert für alle vier Parameter des `frame` Attributs zugewiesen werden kann.

### 5.4.2 Intrinsic Content Size

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

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

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

### 5.4.3 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. 59, Abb. 5.6).

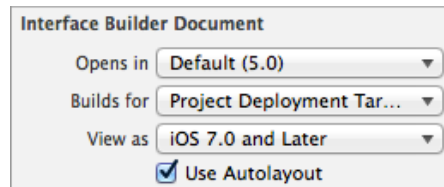


Abbildung 5.6: Auto Layout kann auf Basis einzelner Interface Builder Dateien ein- oder ausgeschaltet werden

Ist Auto Layout eingeschaltet, können wir Constraints auf verschiedene Weise erstellen. Eine Möglichkeit besteht in der Verwendung der Schaltflächen am unteren rechten Rand des Editorbereichs (s. S. 59, Abb. 5.7). Hier finden wir die interaktiven Menüs *Align* und *Pin*, mit denen den ausgewählten Views Constraints hinzugefügt werden können.



Abbildung 5.7: Die Schaltflächen am unteren rechten Rand des Editorbereichs bieten Zugriff auf viele Auto Layout Optionen

Alternativ kann das von IBOutlets und IBActions bekannte Ziehen einer Verbindungslinie bei gehaltener **ctrl**-Taste auch zum Erstellen von Constraints zwischen zwei Interfaceelementen verwendet werden (s. S. 60, Abb. 5.8). 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. 60, Abb. 5.9). Solange das Layout für die View noch uneindeutig ist, werden die Constraints in gelb angezeigt. Eindeutige Layouts werden blau und Layouts mit Konflikten rot markiert.

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. 61, Abb. 5.10). Im *Resolve Auto Layout Issues* Menü am unteren rechten Bildschirmrand ist die Option *Update Frames* zu finden, mit der die Position entsprechend angepasst werden kann.

Alle Constraints einer bestimmten View sind im Size Inspector aufgelistet. Eine Constraint kann wie jedes andere Objekt ausgewählt und mit dem Attributes Inspector bearbeitet werden. Die wichtigsten Optionen sind außerdem mit einem Doppelklick auf eine Constraint im Editorbereich erreichbar (s. S. 61, Abb. 5.11).

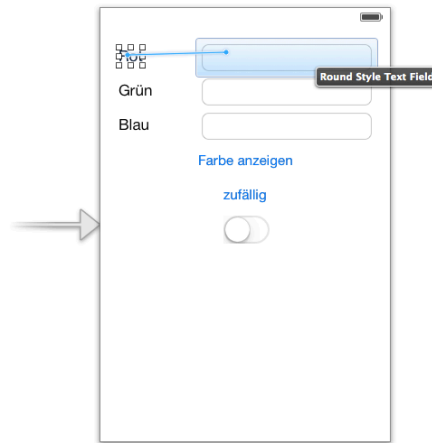


Abbildung 5.8: Constraints können ähnlich wie IBOutlet und IBActions durch Ziehen einer Verbindungslinie erstellt werden

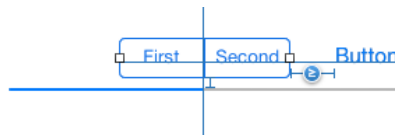


Abbildung 5.9: Ist das Layout einer View durch Constraints eindeutig festgelegt, werden diese blau gekennzeichnet

#### 5.4.4 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:attribute:multiplier:constant:` instanziiert werden. Da jedoch meist mehrere Constraints benötigt werden, stellt `NSLayoutConstraint` zusätzlich eine Klassenmethode `constraintsWithVisualFormat:options:metrics:views:` zur Verfügung, die eine **Visual Format Language** verwendet.

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

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



Abbildung 5.10: Uneindeutige Layouts oder Views, deren Position von vom berechneten Layout abweichen, werden gelb gekennzeichnet und zeigen ihre Position zur Laufzeit als eine gestrichelte Markierung

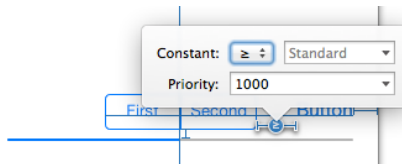


Abbildung 5.11: Ein Doppelklick auf eine Constraint im Interface Builder zeigt die wichtigsten Optionen in einem Popup

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

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

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

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

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

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