Dr. Dirk Koller

Know-how ist blau.

Für iPhone, iPad und iPod touch

iPhone-Apps entwickeln

Applikationen für iPhone, iPad und iPod touch programmieren

> So entwickeln Sie Apps mit dem iPhone SDK und Objective-C

> Xcode, Datenbank-Framework und die wichtigsten Klassen des iPhone OS im Detail

> Entwickeln ist nicht genug: Apps im App Store richtig vermarkten

Von der Idee zum App Store: So realisieren und vermarkten Sie Ihre Apps!

FRANZIS

Dr. Dirk Koller

iPhone-Apps entwickeln

Dr. Dirk Koller

iPhone-Apps entwickeln

Applikationen für iPhone, iPad und iPod touch programmieren

Mit 213 Abbildungen

Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte Daten sind im Internet über http://dnb.ddb.de abrufbar.

Lektorat: Franz Graser
Satz: DTP-Satz A. Kugge, München
art & design: www.ideehoch2.de
Druck: Bercker, 47623 Kevelaer
Printed in Germany

ISBN 978-3-645-60001-9

Inhaltsverzeichnis

Danksagung

Meiner Frau Lisa möchte ich für das Anfertigen der tollen Skizzen danken.

Bei der Durchsicht des Textes haben die Herren Andreas Muth und Claus Weber mitgeholfen. Vielen Dank dafür!

Ein ganz besonderer Dank gebührt meinem Schwiegervater Dr. Josef Hammerschick für die Durchsicht des ganzen Buchs und die Ausführungen zur Entfernungsberechnung.

Zu guter Letzt möchte ich Herrn Franz Graser vom Franzis-Verlag für die guten Tipps und Anregungen sowie die freundliche Zusammenarbeit danken.

1 Einleitung

Willkommen, lieber Leser. Keine Angst, ich werde Ihnen in diesem Kapitel nicht erklären, dass Sie Codefragmente im Buch an einer speziellen Schriftart erkennen können. Trotzdem müssen ein paar einleitende Worte sein. Im Wesentlichen geht es darum, was das iPhone so besonders macht, welche Unterschiede in der Entwicklung bestehen, für wen dieses Buch gedacht ist und was Sie mitbringen sollten, um mit der Entwicklung zu beginnen.

1.1 Das iPhone: ein revolutionäres mobiles Gerät

Vor ein paar Jahren erschien auf *YouTube* [URL-YOUTUBE siehe URL-Verzeichnis im Anhang] eine sehr schöne Persiflage auf die Werbung für Apples neues Wundertelefon namens iPhone. Im Video werden die Fähigkeiten des Geräts ordentlich auf die Schippe genommen. Es wird gezeigt, wie das iPhone als Rasierapparat, Mausefalle, Panflöte, Ultraschallgerät oder Flaschenöffner verwendet werden kann.

Seit dem Erscheinen dieses Videos hat sich einiges getan. Klar, auch heute kann man sich mit dem iPhone (noch) nicht rasieren, dennoch scheinen die Fähigkeiten des Geräts ins Unermessliche zu wachsen. Wer das Filmchen heute betrachtet, gerät bei manchen der verspotteten Anwendungen vielleicht schon ins Grübeln.

Das iPhone als Flöte? Probieren Sie doch einmal die Anwendung *Ocarina* aus.

Das iPhone als Ultraschallgerät? Gerade im Medizinbereich entstehen im Moment dank der geöffneten Hardwareschnittstelle sehr viele Apps.

Darüber hinaus dient das Gerät inzwischen auch als hervorragende Spieleplattform oder als Navigationsgerät.

Die Vision eines mobilen Internets wurde und wird durch das iPhone Wirklichkeit. Weniger aufgrund der Möglichkeit, mit Mobile Safari unterwegs endlich vernünftig browsen zu können, sondern vielmehr durch die unglaubliche Anzahl kleiner Apps, die sich mit dem Internet verbinden, um ihre Dienste anzubieten.

Die neueste Generation von Programmen mischt dabei reale Eindrücke wie den Blick durch die Kamera (etwa auf ein Alpenpanorama) mit virtuellen Informationen aus dem Internet (vielleicht die Namen der Berge mit dazugehörigen Beschreibungen). Erste Vertreter dieser »Augmented Reality«-Sparte, wie beispielsweise *Layar*, bieten einen Ausblick auf die fantastischen Möglichkeiten, die sich durch diese Sinneserweiterung eröffnen.

Ebenso erfolgreich wie das Handy ist Apples neue Softwareverteilungsplattform App Store. Es ist schwer abzuschätzen, ob der App Store so erfolgreich ist, weil iPhone, iPod touch und iPad bei den Anwendern so beliebt sind, oder aber umgekehrt die Existenz

dieser innovativen Softwarequelle die Menschen zum Kauf der zugehörigen Hardware motiviert. Wie auch immer, die Zahl der im App Store verfügbaren Anwendungen liegt inzwischen im sechsstelligen Bereich. Die Anzahl der Downloads geht gar in die Milliarden, und ein Ende des Booms ist nicht abzusehen. Das Konzept wird von nahezu allen großen Smartphone-Anbietern kopiert, und es ist wohl nur noch eine Frage der Zeit, bis auch Desktop-Software im Store verfügbar ist.

Ohne Zweifel hat Apple also mit dem iPhone und dem dazugehörigen Betriebssystem iPhone OS eine attraktive Plattform geschaffen, für die sich Softwareentwicklung lohnt. Ob es sich für Sie in barer Münze auszahlt, kann ich Ihnen leider nicht sagen. Das hängt von vielen Faktoren wie einer tollen Idee, einer gelungenen Umsetzung, einer guten Marketingstrategie, der Zielgruppe usw. ab. Aber zumindest großen Spaß bei der Konzeption und Realisierung Ihrer Anwendung kann ich versprechen.

Wie unterscheidet sich die Softwareentwicklung für das iPhone von der Entwicklung für andere Plattformen?

Die im Vergleich zum Desktop geringe Auflösung des Displays erfordert sauber konzipierte Views, die den vorhandenen Platz effizient und im Sinne der iPhone Human Interface Guidelines nutzen. Neben der Auswahl und Anordnung der User-Interface-Bedienelemente sollte das Navigationsmodell durchdacht und für den Benutzer intuitiv zu verstehen sein.

Die wahrscheinlich größte Hürde besteht für den Entwickler im Verstehen und richtigen Anwenden der Speicherverwaltung. Im Gegensatz zu Mac-OS-Desktop-Anwendungen bietet iPhone OS aus Performance-Gründen keine automatische Garbage Collection. Das Freigeben nicht mehr genutzten Speichers ist Aufgabe des Entwicklers. Kommt er dieser Aufgabe nicht gewissenhaft nach, beendet das Laufzeitsystem die App eventuell recht abrupt.

Eine weitere Besonderheit der Plattform iPhone OS ist die enge Bindung an den (für den Entwickler) oft recht unverständlichen Softwareverteilungsmechanismus von Apple. Schon beim Übertragen der ersten Codierversuche auf das Gerät werden die Nerven des Entwicklers getestet. Nach dem Abliefern der fertigen App in den App Store nehmen selbst hartgesottene Naturen eine Woche Urlaub.

1.2 Für wen ist dieses Buch gedacht?

Dieses Buch richtet sich an erfahrene Entwickler, die mit der iPhone-Entwicklung beginnen möchten oder schon begonnen haben. Erfahrene Entwickler meint hier solche, die mit den Konzepten und der Syntax einer objektorientierten, C-ähnlichen Programmiersprache wie C++, C# oder Java vertraut sind. Anfänger in der Kunst der Programmierung sind willkommen, benötigen aber auf jeden Fall weiteres Material. Für den Einstieg in die Programmierung mit Objective-C (das ist die Sprache, in der Mac- oder iPhone-OS-Anwendungen geschrieben werden) kann »Programming in Objective-C« von Stephen G. Kochan empfohlen werden [KOC09]. Das Kapitel »Objective-C« des vorliegenden Werks erläutert lediglich einige Besonderheiten von Objective-C und ist daher definitiv keine Einführung.

1.3 Die Beispielanwendung

Dieses Buch verfolgt einen sehr praxisnahen Ansatz. Wir (also Sie und ich) begleiten ein »echtes« Projekt vom ersten Konzept über Implementierung, Beta-Test, Debugging usw. bis zur Auslieferung in Apples App Store. Dieser Ansatz bringt es mit sich, dass in unserer App nicht benötigte Frameworks (wie etwa OpenGL oder Core Animation) auch nicht beschrieben werden. Das Buch ist also keine ausführliche Referenz der iPhone-Programmierschnittstelle, sondern beschreibt eher, wie man die kleinen Codestücke aus einer Referenz zu einer echten Anwendung zusammenfügt.

Eine echte Anwendung ist im Gegensatz zu vielen kleinen losgelösten Beispielen eine komplexe Sache. Das Nachverfolgen der Entstehung ist um einiges schwieriger, und der Einstieg ist nicht in jedem Kapitel möglich. Dennoch wird im vorliegenden Buch dieser Ansatz verfolgt, weil nur so der »Klebstoff« zwischen den losgelösten Beispielen vermittelt werden kann. Letztlich ist genau dieser Code der kleine, aber entscheidende Unterschied zwischen einem »Hello World« zu jedem Thema und einer richtigen Anwendung.

Als Beispielanwendung dient eine Zeiterfassung namens *ChronoLog*, die voraussichtlich im App Store erhältlich sein wird. Im Buch wird dafür der Arbeitstitel *chronos* verwendet. Der Sourcecode der ersten Version kann von den Lesern des Buches auf den Webseiten des Autors [URL-KOLLER siehe URL-Verzeichnis im Anhang] kostenlos heruntergeladen werden. Da natürlich nicht immer jede Zeile erklärt werden kann, sollte von dieser Möglichkeit unbedingt Gebrauch gemacht werden.

Die Idee, ein Projekt von Anfang bis zum Ende zu begleiten, hat Auswirkungen darauf, wie das Buch gelesen werden sollte: ebenfalls vom Anfang bis zum Ende. Und zwar möglichst vor dem Rechner mit geöffneter Entwicklungsumgebung. Versuchen Sie, während des Lesens die Beispielanwendung (oder zumindest etwas Ähnliches) nachzubauen. Wenn Sie nicht weiterwissen, werfen Sie einen Blick in den heruntergeladenen Sourcecode.

1.4 Was benötige ich zum Starten?

Falls Sie gänzlich unbedarft von der Windows-Welt in die iPhone-Entwiclung starten, also noch keinen Mac und kein iPhone Ihr Eigen nennen, kommen ein paar Kosten auf Sie zu.

1.4.1 Einen Mac

Die ersten Versuche mit Objective-C kann man notfalls auch unter Windows mit GNUstep [URL-GNUSTEP], einer dem iPhone-SDK ähnlichen Programmierumgebung, unternehmen. Möchte man allerdings wirklich iPhone-OS-Anwendungen entwickeln, ist ein Mac unumgänglich. Die benötigte Software, wie das Software Development Kit (SDK) mit seinen Frameworks und den Entwicklungswerkzeugen, ist nur für Apples Betriebssystem erhältlich. Selbst die bisher einzige Sprachalternative, Mono-

Touch [URL-MONOTOUCH], die eine C#- und .NET-basierte Entwicklung für das iPhone ermöglicht, funktioniert nur auf dem Mac.

Der Entwicklungsrechner muss aber durchaus kein High-End-Produkt der Mac Pro-Klasse sein. Zur Erstellung der Buch-App findet beispielsweise ein Mac mini Verwendung. Einen solchen Rechner können Sie ab ca. 550 Euro erwerben. Natürlich geht mit einem leistungsfähigeren Rechner alles ein bisschen flotter.

1.4.2 Das iPhone-SDK

Das iPhone-SDK ist nach der Registrierung im *iPhone Dev Center* der *Apple Developer Connection* unter [URL-DEVCENTER] kostenlos erhältlich. Als registriertes Mitglied hat man ebenfalls Zugriff auf die umfangreichen Tutorials, Referenzen, Beispiele und vieles mehr.

1.4.3 Ein iPhone, einen iPod touch oder ein iPad

Im Software Development Kit ist ein iPhone-Simulator zum Testen der App enthalten. Wenngleich der Simulator gute Dienste leistet und zum täglichen Entwicklungswerkzeug gehört, kommt man schon bei der Entwicklung nicht um ein echtes mobiles Gerät herum. Der Simulator simuliert eben nur und verhält sich in vielen Situationen nicht wie die echte Hardware. Das Testen aller größeren Entwicklungsabschnitte auf dem Gerät ist unumgänglich.

1.4.4 Eine Entwicklerlizenz

Um die App schon während der Entwicklung auf ein iPhone zu bringen oder sie endgültig in den App Store einzustellen, ist eine Mitgliedschaft in Apples *iPhone Developer Program* Voraussetzung. Der Link hierzu findet sich auch auf den Seiten der *iPhone Developer Connection*. Die Mitgliedschaft im *iPhone Developer Standard Program* kostet 99 US-Dollar und bietet folgende Leistungen:

- Zugriff auf Ressourcen wie das aktuelle SDK, Dokumentation oder Beispielanwendungen,
- Möglichkeit der Erstellung von Development-Profilen (für Entwicklertests) oder Ad-hoc-Profilen (für Betatests),
- technischen Support,
- Zugriff auf die Apple Developer-Foren,
- Möglichkeit der Veröffentlichung im App Store.

Die Mitgliedschaft im teureren Enterprise-Programm ist für größere Firmen gedacht, die eine Verteilung von Apps innerhalb ihres Unternehmens und nicht über den App Store wünschen.

Teil 1 – Die Voraussetzungen

Im ersten Teil des Buches werden die Voraussetzungen geschaffen, um mit der iPhone-Entwicklung beginnen zu können. Dieser Teil ist naturgemäß etwas theorielastig. Auch wenn Sie lieber gerne direkt mit der Entwicklung beginnen möchten, sollten Sie sich unbedingt die Zeit nehmen, um die Kapitel zu studieren. Mangelndes Know-how in komplexen Themen wie der Speicherverwaltung wird sich später rächen.

2 Das iPhone OS

iPhone OS ist ein von Apple entwickeltes Betriebssystem für mobile Geräte wie iPhone, iPod touch oder iPad. Die darin enthaltenen Technologien und Frameworks entsprechen weitgehend denen von Apples Desktop-Betriebssystem Mac OS X. Grundlegende Unterschiede bestehen insbesondere im Bereich des User Interface. Das kleine Display von iPhone und iPod touch und die innovative Multi-Touch-Bedienung erforderten neue Softwarelösungen.

Nach der Vorstellung der enthaltenen Technologien und Frameworks werden drei sehr wichtige Design Patterns besprochen, die bei der Verwendung der Frameworks bekannt sein müssen.

2.1 Technologien & Frameworks

Apple unterteilt die im iPhone OS enthaltenen Technologien in verschiedene Schichten:

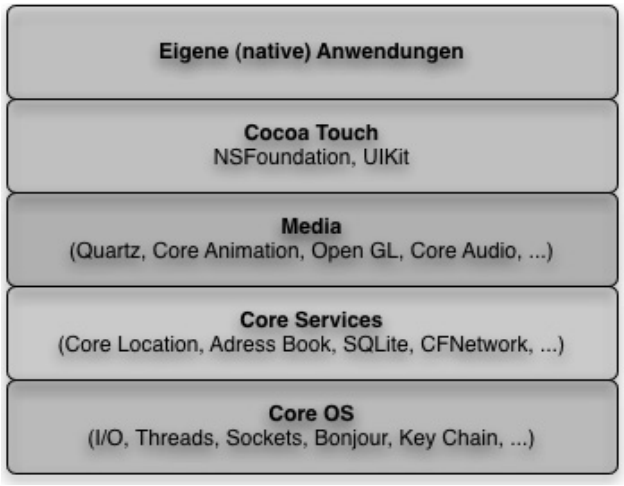


Bild 2.1: Die Architektur des iPhone OS

Die unteren Schichten sind Low-Level-Dienste, die die Grundlage für alle Anwendungen bilden. Weiter oben im Stapel finden sich ausgeklügeltere Funktionen und objektorientierte Abstraktionen für die Basisschichten.

Grundsätzlich sind alle Schichten für den Entwickler zugänglich. Allerdings empfiehlt es sich, »von oben nach unten« zu programmieren, das heißt, sofern möglich, die von Apple entwickelten und gut getesteten Komfortfunktionen der obersten Schicht zu nutzen.

Im Folgenden wird der Inhalt der einzelnen Schichten kurz vorgestellt. Betrachten Sie die Auflistung als ersten Überblick. Die meisten der Frameworks werden im weiteren Verlauf ausführlich besprochen.

2.1.1 Cocoa Touch

Für den iPhone-Entwickler ist die Cocoa-Touch-Schicht am wichtigsten. Hier befindet sich ein Großteil der für die Entwicklung benötigten Funktionalität. Ein besonders wichtiger Bestandteil ist das für mobile Geräte komplett neu entwickelte UIKit-Framework.

Apple Push Notification Service

Aus Performancegründen können auf dem iPhone keine eigenen Hintergrundprozesse verwendet werden. Alle Prozesse werden zusammen mit der zugehörigen App gestoppt. Diese Tatsache sorgte für einigen Unmut bei den Entwicklern. Als Ersatz hat Apple mit SDK 3.0 den Push Notification Service eingeführt. Auch wenn eine Anwendung nicht läuft, können die Benutzer damit über neu verfügbare App-Inhalte (wie etwa eine neue Nachricht) informiert werden. Zur Realisierung der Funktionalität wird eine Serverkomponente benötigt, die über den Apple Push Notification Service Benachrichtigungen verschickt.

AddressBook UI Framework

Das Framework gestattet den Zugriff auf das Standard-User-Interface zur Kontaktanzeige und -bearbeitung. Im Buch wird es in einem eigenen kurzen Kapitel behandelt.

In App Email

Seit der Version 3.0 des iPhone SDK können dank des Message-UI-Frameworks E-Mails auf einfache Weise direkt aus der App versendet werden. Vorher musste dazu zur Mail-App gewechselt werden. Im Kapitel »Zugriff aufs Internet« wird von dieser Funktionalität Gebrauch gemacht.

Map Kit Framework

Das Map Kit Framework gestattet das einfache Einbetten einer (Land-)Karten-Komponente in eigene Anwendungen. Es können Markierungen wie beispielsweise die eigene Position hinzugefügt werden. Auch dieses Framework wird in einem eigenen Kapitel besprochen.

Peer-to-Peer-Support

Die auf der Apple-Technologie *Bonjour* basierende Peer-to-Peer-Funktionalität erlaubt die direkte Kommunikation zwischen zwei mobilen Endgeräten. Insbesondere für Multiplayer-Spiele ist das ein oft benötigtes Feature.

UIKit-Framework

Das UIKit ist das Gegenstück zum AppKit-Framework bei der Mac-Desktop-Entwicklung. Es ist zuständig für alles, was mit der grafischen Darstellung der Applikation zusammenhängt. Im Einzelnen sind dies die folgenden Aspekte:

- User-Interface
- Multi-Touch-Verhalten
- Screen-Rotation
- View Controller
- Events
- Gesten

Das Klassendiagramm in der folgenden Abbildung soll einen ersten Überblick über die verwendeten Klassen und ihre verwandtschaftlichen Beziehungen vermitteln. Wie man sieht, beginnen alle Klassennamen des Frameworks mit dem Präfix `UI` (für User Interface). Beachtenswert sind die Zweige, die durch die Basisklassen `UIView`, `UIViewController` und `UIControl` gebildet werden. Im weiteren Verlauf des Buches werden wir uns ausführlich mit den einzelnen Klassen des UIKit-Frameworks beschäftigen.

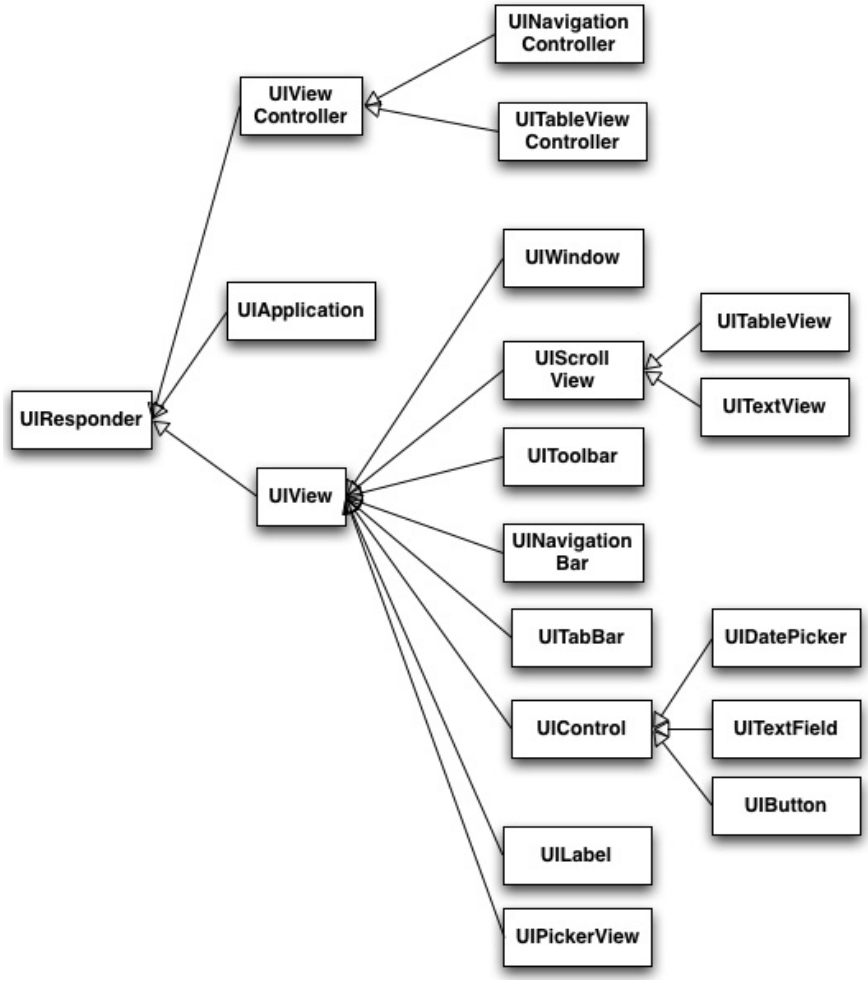


Bild 2.2: Ausschnitt aus der UIKit-Klassenhierarchie

2.1.2 Media

Der *Media-Layer* bietet Grafik-, Audio- und Videotechnologien. Im weiteren Verlauf dieses Buches werden sie keine große Rolle mehr spielen. Unser Fokus liegt eher auf datenzentrierten Anwendungen. Dennoch sollen die verfügbaren Frameworks zumindest kurz angesprochen werden. Für alle Technologien existieren entsprechende Guides und Referenzen im iPhone Developer Center.

Quartz

Quartz ist eine Bibliothek für vektorbasiertes Zeichnen. Es bietet die üblichen Grafikfunktionen für das Erstellen von Linien, Farben, Gradienten usw. Auch PDF-Dokumente lassen sich mit dieser API erstellen.

Core Animation

Animation ist bereits in vielen Komponenten von UIKit enthalten. Der Entwickler hat an einigen Stellen die Möglichkeit, ein Animated Flag auf `YES` zu setzen, um Aktionen animiert durchzuführen. Neben diesen vorgesehenen Systemanimationen können aber auch eigene Ideen für visuelle Effekte realisiert werden.

OpenGL ES

Mit *Open GL* können sprach- und plattformunabhängig schnelle 2-D- und 3-D-Grafiken erstellt werden. Insbesondere Spiele machen von der hardwarenahen API Gebrauch. Der Namenszusatz ES steht für »Embedded Systems«.

AV Foundation

Dieses High-Level-Framework wird zum einfachen Aufnehmen und Abspielen von Audio-Inhalten mit Objective-C benutzt.

Core Audio

Core Audio bietet ebenfalls Funktionen für den Audio-Bereich an, ist aber breiter angelegt. Mit dem C-Framework können Sounds auch bearbeitet und abgemischt werden.

Open AL

Noch eine Audio-Bibliothek, allerdings mit anderer Zielrichtung. *Open AL* erlaubt das Positionieren und Bewegen von Tonquellen in einem dreidimensionalen Raum. Wie Open GL wird die Cross-Plattform-Bibliothek in erster Linie für Spiele benutzt.

Video

Das *Media Player Framework* erlaubt das Abspielen von Videos in verschiedenen Formaten wie `.mov`, `.mp4`, `.m4v`, und `.3gp`.

2.1.3 Core Services

Die *Core Services*-Schicht beinhaltet grundlegende Systemservices, die alle Anwendungen direkt oder indirekt verwenden. Einige der Services werden in separaten Kapiteln gesondert behandelt.

Adressbuch

Das *Adressbuch*-Framework gestattet den Zugriff auf die Daten in der Kontakte-App. Der Zugriff kann lesend oder schreibend erfolgen.

Core Data

Während vor dem SDK 3.0 für Datenbank-Zugriffe die sehr rudimentäre SQLite-API genutzt werden musste, steht seit Version 3.0 das leistungsfähige Persistenz-Framework *Core Data* zur Verfügung. *Core Data* wird später ausführlich in einem eigenen Kapitel behandelt.

Core Foundation

Das *Core Foundation*-Framework ist ein in C implementiertes Framework für die grundlegende Datenverarbeitung. Zu den Schwerpunkten gehören die folgenden Themengebiete:

- Collection-Datentypen (Arrays, Sets usw.)
- Bundles
- Strings
- Datum und Zeit
- Raw Data
- Preferences
- URL und Streams
- Threads
- Port- und Socket-Kommunikation

Bei der iPhone-Entwicklung wird nicht direkt die *Core Foundation,* sondern das Foundation-Framework benutzt. Es bietet dieselbe Funktionalität in Form von Objective-C-Wrappern an.

Core Location

Zur Positionsbestimmung via WLAN, Mobilfunk oder GPS steht das Framework *Core Location* zur Verfügung. Die für mobile Geräte natürlich besonders interessanten Möglichkeiten dieser API werden in einem eigenständigen Kapitel behandelt.

Foundation-Framework

Das *Foundation-Framework* enthält grundlegende Basisklassen für die Mac- oder iPhone-Entwicklung. Sie erlauben unter anderem das Erstellen und Bearbeiten von Nummern, Strings und Collections.

Das folgende Klassendiagramm gibt einen Überblick über die wichtigsten im Buch verwendeten Foundation-Klassen:

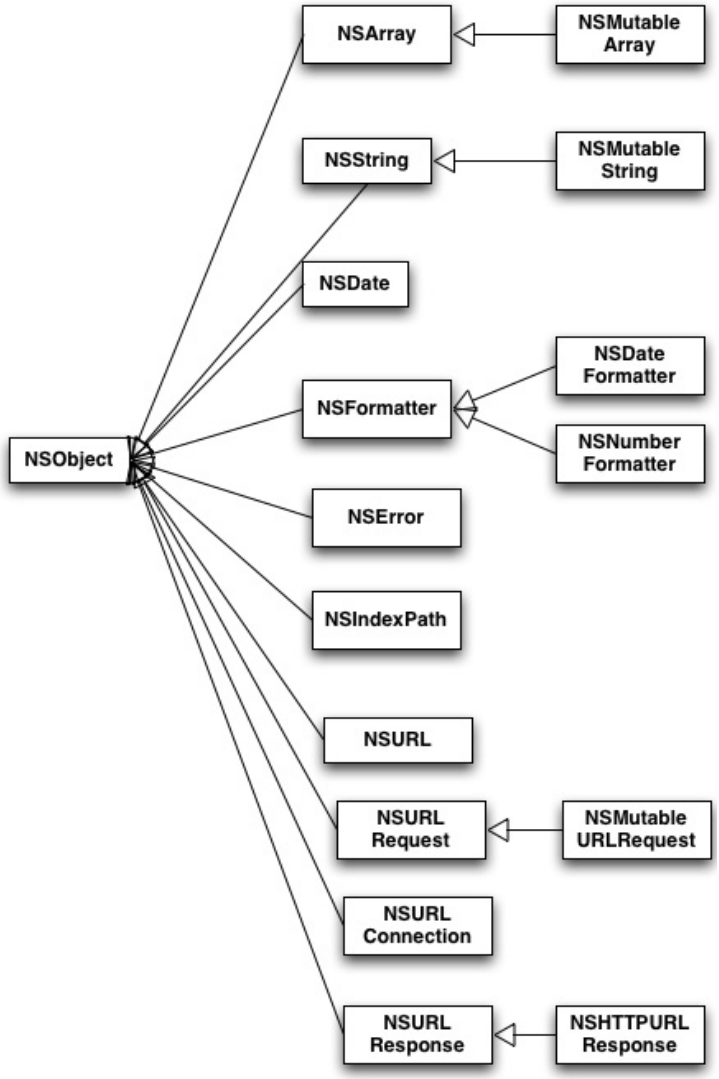


Bild 2.3: Ausschnitt aus der Foundation-Klassenhierarchie

Die Gesamtzahl aller Klassen des Frameworks ist noch deutlich größer. Wie man sieht, beginnen alle Klassen des Foundation-Frameworks mit dem Präfix `NS` (das kommt von »NextStep« und hat historische Gründe) und erben von der Basisklasse `NSObject`.

In App Purchase

Das *Store Kit Framework* erschien ebenfalls mit dem SDK 3.0. Seitdem können aus der Anwendung heraus Zusätze zur App, wie zum Beispiel Landkarten, weitere Level, andere

Vorlagen usw., gekauft werden. Vor dem Erscheinen des Features war etwa ein Anbieter eines Reiseführers gezwungen, eine eigene App für jedes Land herauszubringen. Dadurch wuchs die Zahl der Anwendungen ungeheuer an, und die Übersichtlichkeit blieb auf der Strecke.

SQLite

SQLite ist eine kleine SQL-Datenbank, die auf dem iPhone wie auch auf anderen Smartphones enthalten ist. Zur Datenbank existiert eine C-Library, die, wie bereits erwähnt, nicht den Komfort von *Core Data* bietet. Dennoch kann sie zum Beispiel aus Performancegründen eine interessante Alternative zu dem leistungsfähigeren, aber damit auch schwergewichtigeren Persistenz-Framework sein.

XML-Unterstützung

Zur Verarbeitung von XML kann der eventbasierte XML-Parser *NSXMLParser* genutzt werden. Der Cocoa-DOM-Parser *NSXMLDocument* steht (vermutlich aus Performance-Gründen) nicht zur Verfügung. Eine Alternative ist die Library *libxml2*. Wenn auch XML wegen seiner guten Lesbarkeit Vorteile hat, so sollte man gerade auf dem iPhone über ein leichtgewichtigeres Datenaustauschformat wie JSON (JavaScript Object Notation) nachdenken. Mehr dazu im Kapitel »Zugriff aufs Internet«.

2.1.4 Core OS

CFNetwork

CFNetwork ist ein C-basiertes Framework für Netzwerkzugriffe. Gegenüber den hoch abstrahierten Klassen des Foundation-Frameworks, die im Kapitel »Zugriff aufs Internet« verwendet werden, bietet die Bibliothek bodenständigere Funktionen, mit denen zum Beispiel BSD-Sockets benutzt werden können.

Accessory Support

Seit SDK 3.0 kann über das *Accessory*-Framework mit angeschlossenen Hardwarekomponenten kommuniziert werden.

Security

Das *Security*-Framework erlaubt den Umgang mit Zertifikaten, Schlüsseln und Policies.

System

Hier finden sich der Kernel, zugehörige Treiber und sehr grundlegende UNIX-Interfaces des Betriebssystems.

2.2 Cocoa Design Patterns

Die Verwendung bewährter Entwurfsmuster (Design Patterns) bei der Lösung komplexer Probleme kann ein wichtiger Baustein für ein erfolgreiches Softwareprojekt sein. Allerdings erfordern das Erkennen der Problemsituation und auch die richtige Implementierung des Patterns einige Erfahrung. Cocoa (bzw. das enthaltene Framework *UIKit*) ist für die Verwendung im Sinne einiger Patterns konzipiert, sodass eine Entwicklung ohne diese praktisch nicht möglich (zumindest aber ziemlich fragwürdig) ist. Die wichtigsten Muster werden im Folgenden besprochen. Wer sich weiter mit dem Thema beschäftigen möchte, dem seien der Klassiker *Entwurfsmuster* [GHJV01] sowie der Cocoa Fundamentals Guide, Abschnitt »Cocoa Design Patterns« aus der Apple iPhone OS Library empfohlen.

2.2.1 Model View Controller

Das Model View Controller Pattern ist eines der populärsten Muster überhaupt und fehlt in fast keinem Buch über Softwareentwicklung. Kerngedanke ist die Aufteilung des Codes in verschiedene funktionale Bereiche (eben Model, View und Controller). Das Model repräsentiert das zugrunde liegende Datenmodell. Der View ist zuständig für die Visualisierung der Daten und nimmt Nutzereingaben über Buttons, Textfelder usw. entgegen. Model und View kommunizieren nicht direkt miteinander, sondern über den Controller. Dieser »kennt« (über Instanzvariablen) sowohl das Datenmodell als auch den View. Er liefert dem View die Daten aus dem Model zur Anzeige und ändert die Daten als Reaktion auf entsprechende Aktionen. Die folgende Abbildung zeigt die Zusammenhänge:

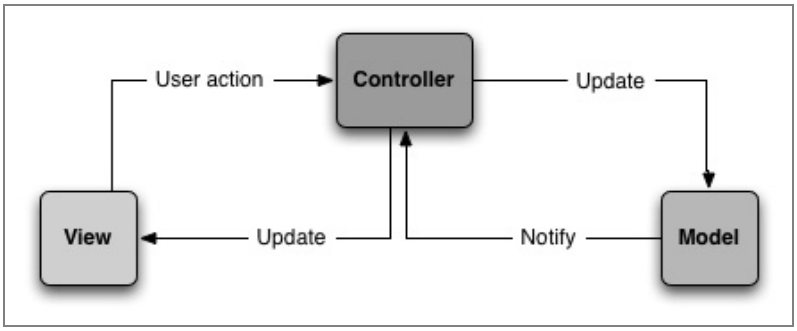


Bild 2.4: Das MVC-Pattern

Das MVC-Pattern findet sich insbesondere in den Klassen `UIView`, `UIViewController` und ihren Subklassen wieder.

2.2.2 Delegation (Delegate)

Delegation kann als Alternative zur Erstellung von Subklassen betrachtet werden. Wie bei der Vererbung erlaubt das Pattern die Erweiterung der Funktionalität einer Klasse. Verantwortlich für die Realisierung der zuzufügenden Funktionalität ist eine weitere Klasse, das Delegate. Die Klasse, die erweitert werden soll, hält eine Instanzvariable für das Delegate und leitet Anfragen für die neue Funktionalität (also Methodenaufrufe) an das Delegate weiter. Sie *ist* also nicht vom Typ des Delegates (wie es bei der Vererbung der Fall ist), sondern sie *hat* ein Delegate.

Die Delegate-Klasse implementiert ein Protokoll, damit klar definiert ist, welche Aufgaben sie eventuell erledigen kann. Die Methoden des Protokolls sind alle optional. Ein Delegate kann also alle oder keine oder einige Methoden des Delegate-Protokolls implementieren. Ob ein Delegate wirklich eine Implementierung für eine bestimmte Methode zur Verfügung stellt, wird über die Methode `respondsToselector` mittels Introspektion ermittelt. Wenn dabei keine Implementierung gefunden wird, wird die Methode im Delegate nicht aufgerufen und stattdessen ein Default-Verhalten ausgeführt. Die Methoden in einem Delegate-Protokoll gehören thematisch zusammen. Eine Klasse kann gegebenenfalls mehrere Delegates für verschiedene Aufgaben haben.

Bei der iPhone-Entwicklung begegnet dem Entwickler das Delegation-Pattern zuerst in Form des *ApplicationDelegate* (dem Delegate des für die gesamte Applikationssteuerung verantwortlichen Application-Objekts), aber auch an zahlreichen anderen Stellen wie beispielsweise dem Erstellen von Textfeldern.

2.2.3 Target Action

Als Target Action Pattern oder Target-Action-Mechanismus wird die Verbindung zwischen einem Bedienelement, etwa einem Button, und der bei Betätigen des Bedienelements aufzurufenden Methode bezeichnet. Das alleinige Auswerten des Hardware-Events nach dem Button-Klick ist unkomfortabel, deshalb wird der Button mit einer Aktion verbunden. In Java realisiert man diese Funktionalität über `ActionListener`. Das Target ist das Objekt, das bei Auftreten des Ereignisses informiert wird. Die Aktion ist die Nachricht, die das Target gesendet bekommt. Das Target ist im Allgemeinen eine vom Benutzer erstellte Klasse, wie beispielsweise ein `ViewController`. Wir werden später sehen, dass die Verbindung zwischen Bedienelement und Target sogar grafisch als Linie gezogen wird.

3 Objective-C

Wie einleitend erwähnt, kann und will dieses Kapitel keine Einführung in Objective-C sein. Wer eine solche sucht, sei hier nochmals auf das lesenswerte Buch *Programming in Objective-C* hingewiesen [KOC09]. Eine gründliche Behandlung der Programmiersprache würde den Umfang dieses Buches sprengen. Leser, die eine objektorientierte Sprache beherrschen, werden aber einen Großteil der Konzepte wie Kapselung, Vererbung oder Interfaces bereits kennen und somit wiederfinden. Der Fokus dieses Kapitels soll deshalb auf den Besonderheiten von Objective-C liegen. Zum Vergleich wird, ihrer weiten Verbreitung wegen, die Sprache Java herangezogen.

3.1 Herkunft der Sprache

Objective-C (oder auch ObjC) ist eine C-Erweiterung, die von Brad J. Cox und Tom Love in den frühen 80er-Jahren entworfen wurde. Wie viele objektorientierte Programmiersprachen lehnt sich Objective-C an SmallTalk an. Die Sprache wurde 1988 von der – von Steve Jobs gegründeten – Firma NeXT lizensiert, die eine Reihe von Programmierbibliotheken sowie das Betriebssystem NEXTSTEP damit entwickelte. NeXT wurde 1996 von Apple übernommen, sodass Sie sich nun mit der Sprache und den Bibliotheken herumschlagen müssen ...

3.2 Nachrichten

Eine für den Einsteiger sehr verwirrende Eigenschaft von Objective-C ist die Klammer-Syntax für das Versenden von Nachrichten an Objekte (im Folgenden vereinfachend als Methodenaufruf bezeichnet). Der einfachste denkbare Aufruf sieht folgendermaßen aus:

```
[window makeKeyAndVisible];
```

Das Objekt `window` erhält die Nachricht `makeKeyAndVisible`, mit anderen Worten: Die Methode `makeKeyAndVisible` des Objekts `window` wird aufgerufen. Um ganz genau zu sein, ist es eigentlich so, dass ein Sender die Nachricht an den Empfänger schickt, woraufhin der Empfänger (das Objekt `window`) nach einer passenden Methode sucht. Nachricht und Methode sind also verschiedene Paar Schuhe.

In Java würde das Ganze so aussehen:

```
window.makeKeyAndVisible();
```

Interessanter wird es, wenn Argumente übergeben werden sollen, wie im folgenden Beispiel das Objekt `aView`:

```
[window addSubview:aView];
```

Das Argument wird hinter einem Doppelpunkt aufgeführt.

Ihre wahre Stärke spielt die Methodennotation erst aus, wenn mehrere Argumente benötigt werden. In den meisten Sprachen werden die Argumente nämlich einfach durch Kommas getrennt. Dabei geht die Information verloren, welche Bedeutung das Argument hat und an welcher Stelle es stehen muss. Zur korrekten Formulierung des Aufrufs wechselt der Entwickler mehrmals zwischen Definition und Aufruf hin und her (symptomatisch ist hier die Frage »Was muss jetzt an die dritte Stelle?«). In Objective-C dagegen steht vor einem Argument stets ein Label zur Beschreibung des folgenden Wertes:

```
[location initWithLatitude:35 longitude:45];
```

Vergleicht man das mit einem Methodenaufruf in Java, werden die Vorteile der Syntax schnell klar:

```
location.init(35, 45);
```

Zugegebenermaßen hätte man die Java-Methode auch `initWithLatitudeAndLongitude` nennen können. Trotzdem ist der Vorteil der benannten Argumente wohl erkennbar.

Natürlich können Methodenaufrufe geschachtelt werden. Prominentes Beispiel hierfür ist die Abfolge der Allokation und Initialisierung von Objekten:

```
dateFormatter = [[NSDateFormatter alloc] init];
```

Hier wird zunächst eine Instanz von `NSDateFormatter` durch den Aufruf der Klassenmethode `alloc` erzeugt und danach die Methode `init` für die Instanz aufgerufen.

Empfänger einer Nachricht können neben einem Objekt oder einer Klasse auch `self` oder `super` sein. `self` entspricht dem Java-Konstrukt `this` und ist ein Verweis auf das Objekt, das die aktuelle Methode ausführt. Bei Verwendung von `super` wird mit der Suche nach einer passenden Methode in der Superklasse begonnen.

3.3 Import

Objective-C kennt eine Reihe von Präprozessor-Direktiven. Dabei handelt es sich um Anweisungen, die vor dem Übersetzen des Programms vom Präprozessor ausgeführt werden. Die Direktiven beginnen immer mit einem »#«-Zeichen. Beispiele sind `#include`, `#if` und `#define`. Eine Liste gängiger Präprozessor-Direktiven mit Erläuterung findet sich im Anhang. `#import` wird an dieser Stelle näher erläutert, weil die Direktive wichtig für das Verständnis der Codebeispiele ist.

Sowohl `#import` als auch die im Anhang aufgeführte Direktive `#include` erlauben, bereits bestehenden Code in eigenen Anwendungen zu verwenden. Für das Einbinden

von Objective-C-Code wird `#import` benutzt. Dadurch wird sichergestellt, dass der Code nur einmalig eingebunden wird.

Ist der Name des Header-Files in doppelten Anführungsstrichen gegeben, sucht der Präprozessor nach der Datei zuerst in dem Verzeichnis, in dem auch die Datei mit der `#import`-Anweisung liegt, also im Arbeitsverzeichnis. Es handelt sich folglich um selbst geschriebene Klassen:

```
#import "MyAppDelegate.h"
```

Ist der Name des Header-Files dagegen in spitzen Klammern gegeben, erfolgt die Suche in speziellen System-Header-File-Verzeichnissen (zum Beispiel `/usr/include` unter Unix). Diese Notation findet Verwendung beim Einbinden von Code aus bestehenden Bibliotheken:

```
#import <objc/Object.h>
```

`Object.h` aus dem Beispiel oben findet man im Dateisystem unter `/usr/include/objc/Object.h`.

3.4 Interface und Implementation

Klassen werden in Objective-C in zwei Dateien, der Header-Datei (Interface) und der Implementierungsdatei (Implementation), beschrieben.

3.4.1 Interface

Im Interface mit der Endung `.h` werden öffentliche Methoden und Instanzvariablen deklariert. Zusätzlich wird hier definiert, von welcher Superklasse eine Klasse erben soll. Mehrfachvererbung ist nicht möglich. Verwendet man das Foundation-Framework, ist die »Mutter aller Klassen« `NSObject`. Mindestens das dort definierte Verhalten erbt jede Klasse. Schauen wir uns ein einfaches Beispiel an:

```
#import <UIKit/UIKit.h>

@interface HoltelfinderAppDelegate : NSObject <UIApplicationDelegate> {

   UIWindow *window;
}
+ (NSString*)host;

@property (nonatomic, retain) IBOutlet UIWindow *window;
@end
```

Die `import`-Direktive wurde bereits besprochen. `UIKit.h` ist eine Art Sammel-Header-File, in das die anderen Header-Files der UIKit-Klassen importiert werden. Die Klasse `HoltelfinderAppDelegate` erbt lediglich von der Superklasse `NSObject` (wird hinter einem Doppelpunkt aufgeführt) und implementiert das Protokoll `UIApplicationDelegate`.

Protokolle entsprechen Java-Interfaces. Sie stehen, gegebenenfalls durch Kommas getrennt, in spitzen Klammern hinter dem Namen der Superklasse.

Innerhalb der geschweiften Klammern werden die Instanzvariablen deklariert, hier eine Variable mit Namen `window` vom Typ `UIWindow`. Der Stern (konventionsgemäß direkt vor der Variablen) kennzeichnet `window` als Pointer. Die Variable enthält also nicht das Objekt selber, sondern eine Adresse, an der das Objekt im Speicher gefunden werden kann. Sie ist also ein Zeiger (»Pointer«) auf das Objekt. Variablen für Objekte sind in Objective-C immer Pointer.

Nach der schließenden geschweiften Klammer folgt die Deklaration der Methoden. Im Beispiel ist dies die Klassenmethode `host`, die ein Objekt vom Typ `NSString` als Rückgabewert hat. Genau genommen müsste man eigentlich von einem `NSString`-Pointer sprechen.

Klassenmethoden, also Methoden, für deren Aufruf keine Instanz der Klasse benötigt wird, sind erkennbar am Pluszeichen vor dem Rückgabewert. Bei »normalen« Methoden (Instanzmethoden) findet sich dort ein Minuszeichen.

Beendet wird die Deklaration der Klasse mit `@end`. Die Codezeile beginnend mit `@property` wird in Abschnitt 3.6, »Properties«, eingehend besprochen.

3.4.2 Implementation

In der Implementationsdatei mit der Endung `.m` erfolgt die Implementierung, also das Ausprogrammieren der in der Header-Datei deklarierten Methoden.

Zu Beginn der Implementierungsdatei steht in der Regel der Import der zugehörigen Header-Datei. Da es sich dabei um keine Library handelt, steht der Name in doppelten Anführungszeichen (und nicht in spitzen Klammern).

Superklassen oder implementierte Protokolle finden im Implementation-Header keine Erwähnung mehr.

Die eigentliche Implementierung der Methoden erfolgt zwischen den Schlüsselwörtern `@implementation` und `@end`.

Beispiel:

```
#import "HoltelfinderAppDelegate.h"

@implementation HoltelfinderAppDelegate

@synthesize window;

+ (NSString*)host
{
   return host;
}
@end
```

3.5 Datentypen

Neben den bekannten C-Datentypen existieren in Objective-C einige Besonderheiten, welche nachfolgend, in aller Kürze, behandelt werden sollen.

3.5.1 Id

`Id` ist ein Pointer auf ein beliebiges Objekt. `Id` wird immer dann benutzt, wenn man den exakten Typ nicht angeben möchte oder kann. Erst zur Laufzeit wird klar, von welchem Typ das Objekt ist und über welche Methoden es verfügt. Beispiel:

```
- (id)initWithStyle:(UITableViewStyle)style {
```

3.5.2 BOOL

Variablen vom Typ `BOOL` können die Werte `YES` und `NO` annehmen. `BOOL` dient also als Boole'scher Typ. Beispiel:

```
- (void)viewWillAppear:(BOOL)animated {
   [self.tableView reloadData];
}
```

3.5.3 SEL

Methodennamen in Objective-C werden auch als Selektoren bezeichnet. Anhand des Namens wird zur Laufzeit die aufzurufende Methode »selektiert«. Der Datentyp `SEL` kann den Methodennamen aufnehmen. Beispiel:

```
- (BOOL)respondsToSelector:(SEL)aSelector
```

3.5.4 Nil

`Nil` ist ein Objekt vom Typ `Id` und repräsentiert den Null-Wert. Das Senden von Nachrichten an `nil` führt, anders als in Java, zu keiner NullPointerException.

Beispiel:

```
if (cell == nil) {
   cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]
autorelease];
   cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
}
```

3.6 Properties

Properties sind Instanzvariablen mit einer Namenskonvention für die `get-` (Accessor) und `set`-Methode (Mutator). Man beachte, dass in Objective-C, anders als etwa in Java, der Accessor kein »get« vor dem Namen der Property hat. Die Methoden für eine Property `name` heißen also schlicht `name` und `setName`.

Der herkömmliche Weg zum Anlegen von Properties sieht folgendermaßen aus:

1) In der Header-Datei werden Variable sowie zugehöriger Getter und Setter deklariert.
2) In der Implementierung werden `get-` und `set`-Methode implementiert, also mit Leben bzw. Code gefüllt.

Um stupide Schreibarbeit zu ersparen und sowohl die Deklaration als auch die Implementierung der Methoden zu erleichtern, wurden die Hilfsmittel `@property` und `@synthesize` geschaffen. Die Leistungsfähigkeit dieser Konstrukte geht allerdings weit über das aus vielen IDEs bekannte »`generate Accessors`« hinaus.

3.6.1 Deklaration: @property

Durch die `@property`-Anweisung werden Accessoren und Mutatoren deklariert:

```
@property (nonatomic, retain) NSString *name;
```

Die Anweisung ersetzt also:

```
-(void)setName:(NSString*) aName;
-(NSString*)name;
```

Die `@property`-Anweisung steht in der Header-Datei dort, wo auch die Deklaration der Methoden erfolgen würde – also zwischen der schließenden geschweiften Klammer und `@end`.

Die Attribute `nonatomic` und `retain` in der Klammer werden weiter unten behandelt.

3.6.2 Implementierung: @synthesize

Durch die folgende `synthesize`-Anweisung lässt sich die Implementierung der Methoden erzeugen:

```
@synthesize name;
```

Der `synthesize`-Ausdruck steht in der Implementierungsdatei. Genau dort sollen schließlich Getter und Setter erzeugt werden.

Erzeugt wird, abhängig von den noch zu besprechenden Property-Attributen, beispielsweise folgende Implementierung:

```
-(void)setName:(NSString *)aName
{
   name = aName;
```

```
}

-(NSString *)name
{
   return name;
}
```

3.6.3 Property-Attribute

Wie wir schon gesehen haben, können in der runden Klammer hinter `@property` Attribute angegeben werden. Diese beeinflussen die Erzeugung der Methoden.

Readwrite/Readonly

`Readwrite` ist der Default-Wert und führt zur Erzeugung von Getter und Setter. Bei Verwendung von `Readonly` wird die `set`-Methode nicht erzeugt.

Nonatomic

Wenn man sicher ist, dass keine Multithreading-Probleme zu befürchten sind (zum Beispiel, weil man keine neuen Threads erstellt hat), kann durch die Verwendung des Attributs `nonatomic` ein wenig Performance gespart werden. Intern entfällt das Sperren der Methode, während sie von einem Thread abgearbeitet wird.

Assign/Retain/Copy

Die Speicherattribute `assign`, `retain` und `copy` werden abhängig vom Typ der Property vergeben.

`assign` wird für alle Nicht-Objekte verwendet. Im Setter erfolgt bei Verwendung von `assign` eine einfache Zuweisung des übergebenen Wertes.

```
   property = newValue;
```

Handelt es sich bei der betreffenden Property um ein Objekt (im Zweifel erkennbar an dem Stern vor dem Variablennamen), stehen noch `copy` und `retain` zur Verfügung.

Wenn das Objekt das `NSCopy`-Protokoll befolgt, kann `copy` verwendet werden. Ob das der Fall ist, kann der Klassendokumentation entnommen werden. In erster Linie betrifft das Properties vom Typ `NSString`.

Durch das `copy`-Attribut wird in der `set`-Methode vom übergebenen Wert eine Kopie gemacht, und der Pointer der Property-Variable auf die Kopie gebogen. Durch `release` wird zuvor signalisiert, dass der vorherige Wert nicht mehr benötigt wird:

```
   if (property != newValue)
   {
     [property release];
     property = [newValue copy];
   }
```

Für alle anderen (also fast alle) Objekte muss `retain` verwendet werden.

Auch in diesem Fall wird zunächst das Desinteresse am alten Objekt durch `release` bekannt gegeben. Intern wird dadurch ein Zähler verringert. Erreicht der Zähler den Wert 0, wird das Objekt nicht mehr gebraucht und kann gelöscht werden.

Danach wird der Pointer der Property-Variable auf das Objekt im Speicher gesetzt, auf das auch der übergebene Wert deutet. Es zeigen nun zwei Pointer auf das Objekt im Speicher. Damit das Objekt nicht gelöscht wird, wenn einer der beiden kein Interesse mehr hat, muss der Zähler für die Interessenten um eins erhöht werden. Dies geschieht mit `retain`:

```
if (property != newValue)
{
  [property release];
  property = [newValue retain];
}
```

So weit, so gut. Mit diesen Regeln kann man leben. Dummerweise gibt es noch eine wichtige Ausnahme. Wenn sich zwei Objekte gegenseitig referenzieren, würde bei Verwendung von `retain` eine zyklische Abhängigkeit entstehen. Keines der Objekte könnte dann freigegeben werden. In diesem Fall muss eines der Objekte mit `assign` eine sogenannte weak reference erhalten.

Sollten die Unterschiede in der Methodenerzeugung durch die Speicherattribute noch nicht ganz klar sein, ist das gar kein Problem. Im Kapitel »Memory-Management« schauen wir uns die Problematik noch näher an. Merken Sie sich vorerst einfach, dass Properties für Objekte am besten mit `retain` (bzw. mit `copy` im Fall eines `NSString`-Objekts) und für Nicht-Objekte mit `assign` deklariert werden.

Aber Vorsicht! Nicht alles, was für den Java-Umsteiger in Objective-C auf den ersten Blick wie ein Objekt aussieht, ist auch wirklich eins. Bei den primitiven Datentypen wie `int` oder `float` ist die Sache klar. Was aber ist zum Beispiel mit `NSInteger` oder `CLLocationCoordinate2D`?

Der Compiler beschwert sich bei folgender Property-Deklaration mit der Fehlermeldung `"error: property 'number' with 'retain' attribute must be of object type"`.

```
@property (nonatomic, retain) NSInteger *number;    // Vorsicht falsch!
```

`NSInteger` ist keine Klasse, sondern eine plattformabhängige Typdefinition, die zu `long` oder `int` ausgewertet wird. Und auch `CLLocationCoordinate2D` ist kein Objekt, sondern ein *Struct*. Bei Problemen dieser Art schafft ein Blick in die Dokumentation nach der Abmahnung durch den Compiler aber schnell Klarheit.

3.6.4 Punktnotation

Auf Properties, egal ob mit `@property` oder herkömmlich definiert, kann über die Punktnotation (`objekt.property`) zugegriffen werden.

Das funktioniert sowohl für Getter als auch für Setter:

```
person.name;
person.name = @"Maier";
```

Zum Vergleich die zugehörigen konventionellen Aufrufe:

```
[person name];
[person setName:@"Maier"];
```

Beim Verwenden der Punktnotation wird immer die Acessor/Mutator-Methode verwendet (und nicht die Property direkt gesetzt). Betrachten wir als Beispiel den folgenden Ausdruck:

```
person.name = @"Tom";
```

Hier wird nicht etwa die Instanzvariable `name` des Objekts `person` direkt gesetzt, sondern die `set`-Methode aufgerufen. Abhängig von Property-Attributen hat dieser kleine Unterschied große Auswirkungen, zum Beispiel auf das Speicherverhalten.

3.7 Protokolle

Protokolle entsprechen in ihrer Funktionalität in etwa den Java-Interfaces. Sie enthalten Methodendeklarationen, die von Klassen implementiert werden können. Im Gegensatz zu den im Header-File deklarierten Methoden gehören die Methoden im Interface allerdings nicht zu einer bestimmten Klasse, sondern können von mehreren Klassen implementiert werden.

Ein Protokoll wird mittels `@protocol` wie folgt definiert:

```
@protocol MeinProtokoll
- (void)nichtOptionaleMethode;

@optional
- (void)optionaleMethode;

@required
- (void)nochEineNichtOptionaleMethode;

@end
```

Durch die Verwendung der Compiler-Direktiven `@optional` und `@required` können die deklarierten Methoden in optionale und Pflichtmethoden gruppiert werden. Wie im Codebeispiel oben angedeutet, ist `required` Default.

Die Protokolle, die von einer Klasse implementiert werden, sind in spitzen Klammern und durch Kommata getrennt hinter dem Namen der Superklasse im Interface-File aufzuführen:

```
@interface MeineKlasse : Superklasse <MeinProtokoll>
```

Um dem Protokoll zu genügen, sind alle `required`-Methoden im `@implementation`-Teil zu implementieren.

3.8 Kategorien

Kategorien (Categories) werden in diesem Buch nicht verwendet. Da es sich jedoch um ein wichtiges Sprachelement handelt, soll hier trotzdem kurz darauf eingegangen werden.

Mit Categories können Methoden ähnlich wie beim Erstellen einer Subklasse zu existierenden Klassen hinzugefügt werden. Der Clou dabei ist, dass es sich nicht um Klassen aus dem eigenen Projekt handeln muss. Auch fremde Klassen, beispielsweise aus Bibliotheken, können durch eigene Methoden erweitert werden.

Die Definition einer Category sieht aus wie das Interface einer Klasse, lediglich der Name der Category (in Klammern hinter dem Interface-Namen) weist auf die Category hin. Möchte man zum Beispiel der Klasse `NSString` eine neue Methode `specialFormat` zufügen, sieht die Definition der Category `MyFormat` folgendermaßen aus:

```
@interface NSString (MyFormat)
- (NSString *) specialFormat;
@end
```

Die Definition wird in einem File mit der Endung `.h` gespeichert. Der Dateiname sollte sowohl den Namen der erweiterten Klasse als auch den Namen der Category enthalten. Gemäß Konvention steht zwischen den beiden Namen ein »+«, also zum Beispiel `NSString+MyFormat.h`.

Die Implementierung in der Datei `NSString+MyFormat.m` erfolgt wie gewohnt. Auch hier steht der Name der Category in Klammern hinter dem Klassennamen:

```
@implementation NSString (MyFormat)
```

Die neue Methode kann wie eine reguläre `NSString`-Methode verwendet werden.

Das Konzept der Category funktioniert nur für Methoden. Es können keine Instanzvariablen zugefügt werden. Allerdings sind die Instanzvariablen der zu erweiternden Klasse verfügbar.

4 Entwicklungswerkzeuge

Wenn Sie von der Java-Welt zur iPhone-Entwicklung kommen, werden Sie vielleicht überrascht sein, dass Sie nicht die freie Wahl der Entwicklungsumgebung haben. Die Programmierung ist nur mit den Tools von Apple möglich. Die gute Nachricht ist, dass diese Werkzeuge sehr ausgereift sind und die wohl fortschrittlichste Umgebung zur Realisierung mobiler Anwendungen darstellen. Die einzelnen Programme sind sehr gut miteinander verbunden und lassen wenige Wünsche offen. Auch nicht ganz unwichtig ist die Tatsache, dass die komplette Entwicklungsumgebung kostenlos ist.

Die Werkzeuge sind nach einer einfachen Mac OS X-Installation zunächst nicht verfügbar. Am besten laden und installieren Sie das iPhone SDK aus dem iPhone Developer Center [URL-DEVCENTER], um an die aktuellen Versionen der Hilfsmittel zu kommen. Sie finden sie dann nach der Installation unter `/Developer/Applications` auf der Festplatte.

Die Besprechung der Anwendungen bietet einen ersten Überblick und weist auf die in der Praxis am häufigsten verwendeten Funktionalitäten hin. Wer sich tiefer einarbeiten möchte, findet zu allen Programmen den passenden User Guide in der *iPhone Reference Library* (ebenfalls erreichbar über das Developer Center).

4.1 Xcode

Xcode ist Apples IDE, also das Gegenstück zu Eclipse oder NetBeans in der Java-Welt. Xcode wird vorrangig benutzt, um Code zu schreiben, zu kompilieren und zu starten (und natürlich gegebenenfalls zu debuggen). Die Leistungsfähigkeit der IDE geht allerdings weit über diese grundlegenden Tätigkeiten hinaus. Durch die Möglichkeit, auch Datenmodelle in Xcode zu erstellen, und durch die enge Verzahnung mit den anderen Werkzeugen wie Interface Builder oder Instruments steht eine geschlossene und stimmige Entwicklungsumgebung zur Verfügung.

Die ersten Funktionen, die Sie benötigen, sind sicherlich das Anlegen von Projekten und Dateien mithilfe des *File*-Menüs. Falls Sie es noch nicht getan haben, erstellen Sie zum Ausprobieren ein neues iPhone-Projekt (eine View-basierte Applikation) mittels *File > New Project...* Der Assistent erzeugt basierend auf einer Vorlage die komplette Projektstruktur. Wir werden später einen Blick auf die einzelnen generierten Dateien werfen. Neue Dateien können dem Projekt mittels *File > New File…* hinzugefügt werden. Über den Assistenten können grundlegende Eigenschaften wie etwa die Superklasse definiert werden.

Aus dem Menü *Run* soll auf den Eintrag *Run > Debug* zum Starten der Anwendung mit aktivierten Breakpoints sowie auf *Run > Konsole* zum Einblenden der ansonsten schmerzlich vermissten Konsole hingewiesen werden.

Ähnlich gut versteckt wie die Konsole ist ein sehr wichtiges Tool namens *Organizer*. Sie finden es unter *Windows > Organizer*. Der *Organizer* spielt bei der Verbindung zwischen iPhone und Mac eine wichtige Rolle. Wenn Sie iPhone, iPod touch oder iPad über das USB-Kabel mit dem Mac verbinden, sollte das Gerät im Organizer unter *DEVICES* mit einem grünen Punkt versehen sein. Sollte der Punkt gelb oder, schlimmer noch, rot sein, liegt ein Problem vor, das eine Installation der App auf dem Gerät vorerst verhindert.

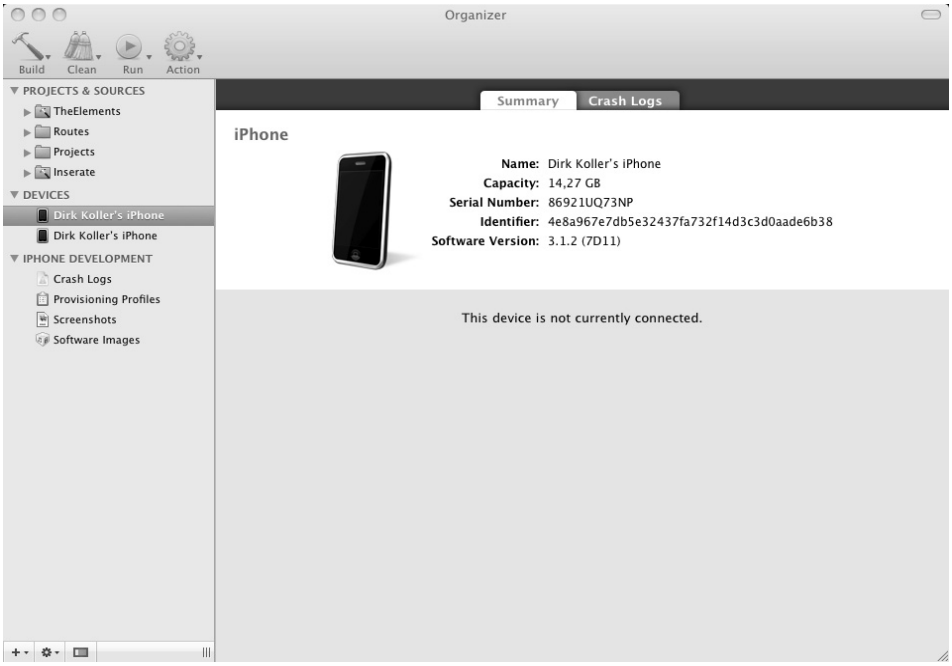


Bild 4.1: Das Organizer-Werkzeug ohne angeschlossenes iPhone

Die sehr gute *Dokumentation* lässt sich durch *Help > Developer Documentation* anzeigen.

Am oberen Fensterrand unterhalb des Menüs findet sich die Toolbar. Die Toolbar ist wie in vielen Anwendungen über das Kontextmenü an die eigenen Bedürfnisse anpassbar. Abhängig von der Xcode-Version ist die Standardbelegung der Buttons leicht unterschiedlich.

Bild 4.2: Die Toolbar in Xcode

Wichtigster Toolbar-Button ist der mit dem Zimmermannshammer gekennzeichnete Knopf zum Erzeugen des Build und zum anschließenden Start der Anwendung. Ob die Anwendung dabei im Simulator oder auf dem (notwendigerweise angeschlossenen) iPhone gestartet wird, kann über das Listenfeld *Overview* auf der linken Seite der Werkzeugleiste gesteuert werden. Je nach installiertem SDK werden hier verschiedene Versionen für Gerät und Simulator angeboten. Einmal laufende Anwendungen lassen sich mit dem roten Button in Form eines Stoppschilds beenden. Der blaue Info-Button öffnet einen Dialog zur Bearbeitung von Einstellungen.

Der Fensterbereich unterhalb der Toolbar ist dreigeteilt:

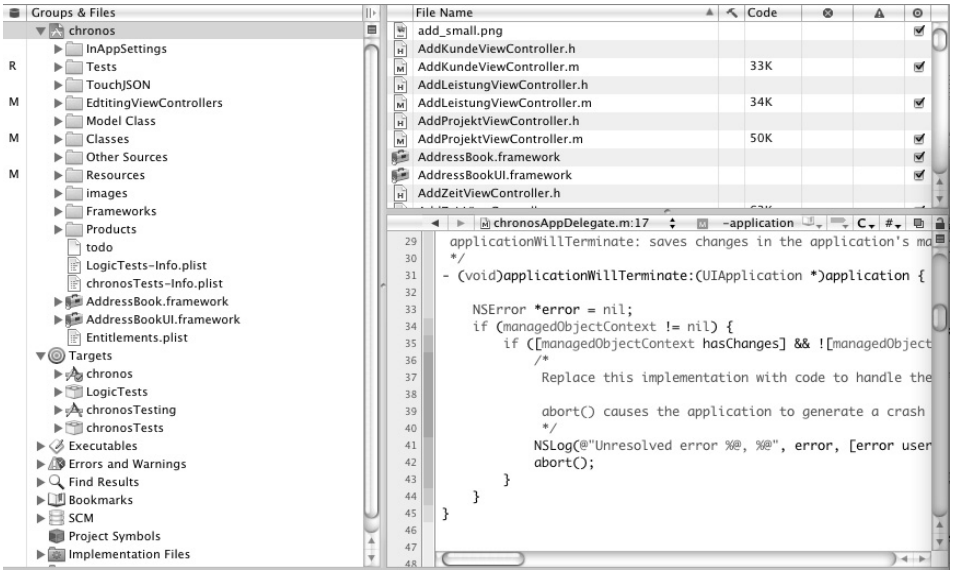


Bild 4.3: Bereiche in Xcode

Im View *Groups & Files,* der sich auf der linken Seite befindet, werden wichtige Informationen zum Projekt in einer Baumstruktur dargestellt. Im Knoten, der den aktuellen Projektnamen trägt, sind die zum Projekt gehörenden Dateien (zum Beispiel Header- und Implementation-Files, Dateien mit User-Interface-Informationen, Property-Listen usw.) aufgelistet. Außerdem finden sich hier die im Projekt verwendeten Frameworks (zum Beispiel UIKit, Foundation, Core Data usw.).

Die Struktur ist unabhängig von der Ablage im Dateisystem und kann zum Beispiel über das Kontextmenü an die eigenen Bedürfnisse angepasst werden.

Die anderen Knoten wie *Target* oder *Executables* sind im Moment nicht ganz so entscheidend und werden bei der Verwendung im weiteren Verlauf besprochen.

Beachtenswert ist das Kontextmenü, das sich bei Rechtsklick auf den Header mit dem Text *Groups & Files* öffnet (siehe folgende Abbildung). Hier können weitere Attribute

der Dateien sichtbar gemacht werden. Standardmäßig wird nur der Source-Code-Management-Status der Datei (beispielsweise A für *Added* oder M für *Modified*) angezeigt.

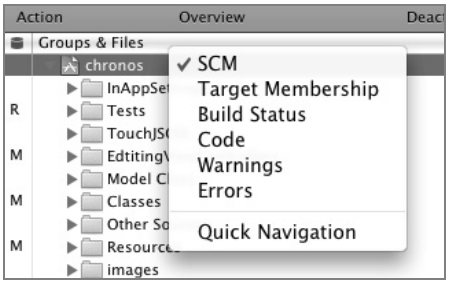


Bild 4.4: Kontextmenü im Header von Groups & Files

Der obere View auf der rechten Seite enthält weitere Informationen zu den auf der linken Seite ausgewählten Objekten. Zumindest bei der Bearbeitung von Programmcode kann gut auf diese Darstellung verzichtet werden. Der View kann über das Menü mit *View > Zoom Editor In* ausgeblendet werden. Der gewonnene Platz steht dann für den Editor darunter zur Verfügung. Der Code-Editor verfügt über die gängigen Features wie *Syntax-Highlighting*, *Code Completion* oder *Code Folding*. Ein Rechtsklick im Editor öffnet das Kontextmenü mit der ganzen Palette der Möglichkeiten. Sehr wichtige Funktionen sind *Jump to Definition* und *Find Text in Documentation*. Diese führen, etwa bei einem markierten Klassennamen, zum Sourcecode bzw. zur Beschreibung der Klasse in der Dokumentation. Die beiden Funktionen benötigt man so oft, dass es sich lohnt, die Tastaturkürzel (APFEL + DOPPELKLICK bzw. Alt + DOPPELKLICK) zu kennen.

Auch im Code-Editor bietet der Header eine Reihe sehr hilfreicher Features. Die beiden Pfeile auf der linken Seite und das darauf folgende Listenfeld gestatten den Zugriff auf die Historie der bearbeiteten Dateien. Die Listbox dahinter bietet einen schnellen Zugriff auf die Methoden der bearbeiteten Klasse. Die Inhalte dieser Box lassen sich zur besseren Übersicht durch die `#pragma mark`-Direktive gruppieren:

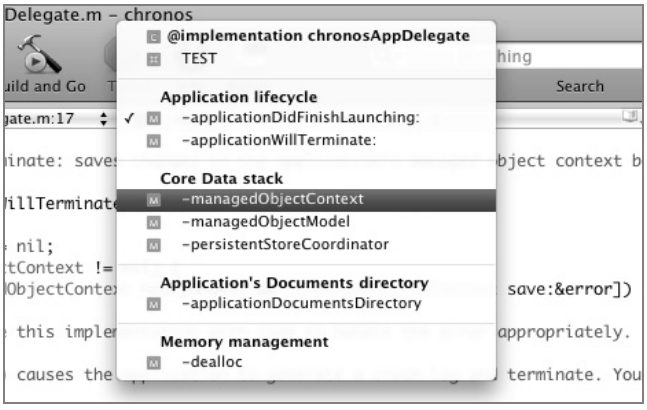


Bild 4.5: Methoden der Klasse im Schnellzugiff

4.2 Simulator

Wurde in Xcode in der Listbox *Overview* der Simulator als *Active SDK* ausgewählt, öffnet sich nach einem *Build & Go* die Anwendung im Simulator.

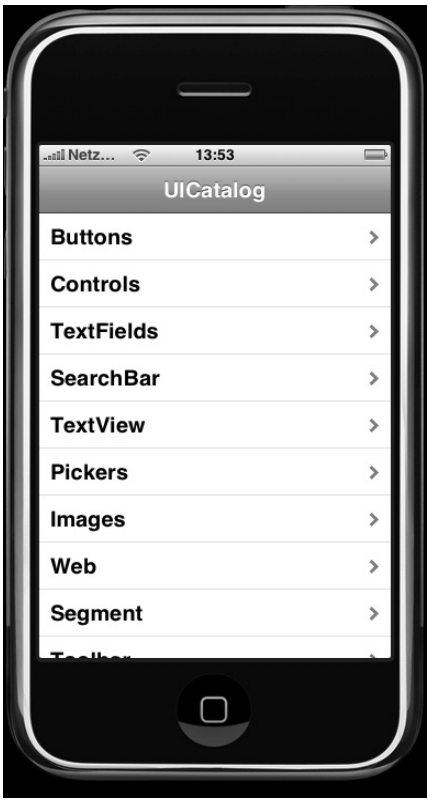


Bild 4.6: iPhone-Simulator

Dieser entspricht in Aussehen und Bedienung dem iPhone, verhält sich aber erfahrungsgemäß in vielen Situationen doch leicht anders.

Grundlegende Unterschiede sind durch die unterschiedliche Hardware bedingt. Während der Software-Stack im Fall eines »echten« iPhones für einen ARM-Prozessor kompiliert wurde, liegt er im Simulator (der auf dem Mac läuft) in einer Intel-Variante vor. Außerdem fehlen dem Mac natürlich einige auf dem mobilen Gerät vorhandene Hardwarekomponenten. So können etwa keine Telefonate mit dem Simulator geführt werden, und die Positionsbestimmung funktioniert nicht. Leider sind zusätzlich einige nicht ganz so offensichtliche Abweichungen vorhanden.

Wie eingangs erwähnt, ist es deshalb nicht zu empfehlen, eine neue Anwendung während der gesamten Entwicklung ausschließlich auf dem Simulator zu testen. Es ist durchaus möglich, dass die App auf dem Simulator läuft und auf dem Gerät Probleme verursacht. Insbesondere beim Suchen nach Memory-Leaks mit Instruments empfiehlt

es sich, mit angeschlossenem Gerät zu arbeiten. So manches beim Test mit dem Simulator aufgespürte Speicherleck löst sich beim Test mit dem echten Gerät in Wohlgefallen auf.

Die Bedienung des Simulators ist weitgehend selbsterklärend. Erwähnenswert sind vielleicht die Funktionen zur Simulation einiger Ereignisse im Menü *Hardware*. Hier kann beispielsweise der Speichernotstand ausgerufen werden, um zu kontrollieren, ob die Methoden zur Behandlung dieses Problems korrekt funktionieren.

Ausgaben wie Log- oder Fehlermeldungen sind bei der Benutzung des Simulators direkt in der Konsole zu sehen.

4.3 Interface Builder

Selbstverständlich können Sie die grafische Oberfläche Ihrer Anwendung durch Schreiben von (sehr, sehr viel) Code in Xcode erzeugen. Deutlich komfortabler ist das Ganze allerdings mit dem Interface Builder. Mit diesem Werkzeug können Controls (wie etwa Buttons und Textfelder) oder Views aus einer Bibliothek per Drag & Drop auf eine Designoberfläche gezogen und anschließend per Property Inspector weiter konfiguriert werden. Um die GUI-Komponenten im Code verfügbar zu machen oder Aktionen (zum Beispiel nach Klick auf einen Button) auszuführen, müssen sie mit Instanzvariablen oder Methoden verknüpft werden. Das fertige User Interface, bestehend aus Objekten und Verbindungen, wird in einer Datei gespeichert und kann in der Anwendung zum Leben erweckt werden.

4.3.1 Xib-Files

Wenn Sie bereits eine View-basierte Applikation generiert haben, finden Sie im Ordner *Resources* Ihres Projekts zwei Dateien mit der Endung `.xib` (`MainWindow.xib` und `{Projektname}ViewController.xib`). Durch einen Doppelklick auf `{Projektname}ViewController.xib` wird die Datei mit dem Interface Builder geöffnet.

Xib-Dateien enthalten die Beschreibung der erstellten GUI im XML-Format. Da diese Dateien vor dem Leopard-Release von Mac OS X im binären Format mit der Endung `.nib` existierten, spricht man oft auch von Nib-Files. Auch jetzt noch werden die umfangreichen XML-Dateien aus Performance- und Platzgründen beim Kompilieren in das binäre `.nib`-Format umgewandelt. Daher wird der Ausdruck »Nib-File« generell als Synonym für Oberflächendefinitions-Datei verwendet. Diese Dateien sollten grundsätzlich nicht von Hand editiert werden. Wer den Dingen gern auf den Grund geht, kann sich die XML-Struktur aber mit einem Texteditor ansehen.

Nach dem Start des Interface Builder präsentieren sich zwei Fenster: das Document Window und die Designoberfläche.

4.3.2 Document Window

Das Document Window stellt eine Art Inhaltsverzeichnis für das Nib-File dar. Alle enthaltenen Interface-Elemente sind hier zugänglich.

In der voreingestellten Icon-Ansicht sind unglücklicherweise nur die Top-Level-Elemente sichtbar:

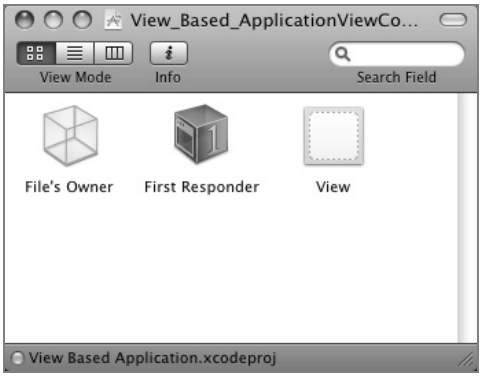


Bild 4.7: Das Document Window

Es empfiehlt sich daher, in der Toolbar des Document Window den View Mode auf Listendarstellung (der mittlere Button) zu ändern. In dieser Ansicht sind weitere Informationen wie der Typ der Elemente und vor allem – nach dem Hinzufügen von weiteren Komponenten – der hierarchische Aufbau erkennbar.

Grundsätzlich lassen sich in Nib-Files folgende Inhalte speichern:

- Objekte
- Proxy-Objekte
- Connections

Bei den in der letzten Abbildung im Document Window dargestellten Objekten *File's Owner* und *First Responder* handelt es sich um Proxy-Objekte. Diese Objekte werden nicht im Interface Builder erzeugt, sie befinden sich nicht im Xib-File. Die Symbole sind Stellvertreter für äußere Objekte. Verbindungen mit diesen Objekten erlauben die Verknüpfung von Nib-Objekten mit Objekten, die in Xcode erzeugt wurden.

File's Owner ist im Normalfall ein View Controller, und zwar derjenige, der das Nib-File geladen hat.

First Responder gestattet den Zugriff auf den ersten Verantwortlichen in der Event-Abarbeitungskette (Responder Chain).

Das Objekt *View* ist kein Stellvertreter, sondern ein echtes Objekt. Ein Doppelklick darauf öffnet den View in der zugehörigen Designoberfläche (falls er noch nicht geöffnet ist).

4.3.3 Designoberfläche

Die Designoberfläche zeigt die Oberfläche, wie sie auch später in der Anwendung aussehen wird. Der Pfeil in der rechten oberen Ecke erlaubt das Drehen der Ansicht um 90°, gestattet also einen Wechsel von der Porträt- in die Landscape-Ansicht. Das ist wichtig, um zu sehen, wie sich die Oberfläche bei gedrehtem iPhone darstellt.

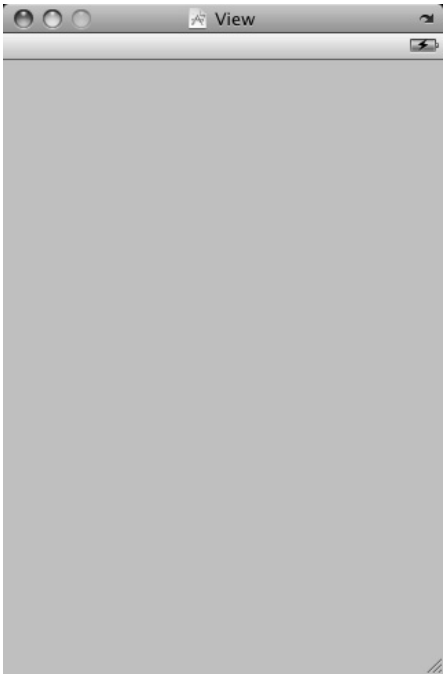


Bild 4.8: View Window

Ein neu erzeugter View enthält natürlich noch keine Komponenten und präsentiert sich als graue Fläche.

4.3.4 Library

Um Komponenten zufügen zu können, müssen Sie via Menü (*Tools* > *Library*) ein weiteres Fenster öffnen, die Komponenten-Library:

Bild 4.9: Interface Builder Library

Das Fenster ist im voreingestellten *Objects*-Reiter dreigeteilt und erlaubt die Auswahl von Komponenten aus einer kaskadierenden Master-Detail-Ansicht. Um unnötiges Suchen zu vermeiden, achten Sie bei den ersten Schritten darauf, immer *Cocoa Touch* in der Baumansicht ganz oben ausgewählt zu haben.

Auch dieses Fenster passt man am besten erst einmal an die eigenen Bedürfnisse an und wählt die Ansicht *View Icons and Descriptions* aus der Listbox mit dem Zahnrad am unteren Fensterrand. Wie der Name schon verrät, werden dadurch Beschreibungen zu den Komponenten angezeigt, die gerade für den Einsteiger ganz hilfreich sind.

Die vorkonfigurierten Controls und Views aus der Library können per Drag & Drop auf die Designoberfläche gezogen werden. Der Interface Builder blendet vor dem Drop allerlei Hilfslinien (»Guides«) ein, um die Ausrichtung der neuen Komponente zu erleichtern.

Für einen ersten Versuch ziehen Sie ein Label aus der Library in das View-Fenster der Datei `ViewController.xib` aus dem Testprojekt. Ein erneuter Start der Anwendung in Xcode sollte das Label im Simulator anzeigen.

Beim Selektieren des Labels im Interface Builder werden Markierungspunkte angezeigt, die ein Vergrößern oder Verkleinern der Komponente erlauben. In diesem Modus kann die Komponente auch verschoben werden. Ein Doppelklick auf das Label gestattet das Ändern des Textes.

Richtig komfortabel ist das Hinzufügen einer weiteren Komponente, zum Beispiel eines Text Field (Textfeld). Hier zeigt der Interface Builder weitere Guides an, die eine Orientierung der Komponente am zuvor hinzugefügten Label erlauben.

In den meisten Fällen werden Sie Komponenten aus der Library in Ihre Views ziehen. Aber auch der umgekehrte Fall ist möglich, denn die Library ist erweiterbar. Durch Auswahl des Knotens *Custom Objects* in der Baumansicht und anschließendes Drag & Drop einer Komponente aus der Designoberfäche in die Library können eigene Komponenten hinzugefügt werden. Das ist sinnvoll für aufwendig konfigurierte Controls, die man wiederverwenden möchte. Analog zu den Standardkomponenten können Name und Beschreibung angegeben werden.

Zum Abschluss der Library-Besprechung sei noch auf den Reiter *Media* am oberen Bildschirmrand hingewiesen. In der Media-Ansicht bietet die Library alle dem Projekt hinzugefügten Ressourcen wie etwa Bilder zur Auswahl an. So können beispielsweise Buttons sehr einfach per Drag & Drop mit Icons versehen werden.

4.3.5 Property-Inspektoren

Ein viertes Fenster namens Property Inspector wird benötigt, um die Komponenten auf dem View weiter konfigurieren zu können. Es ist über das Menü unter *Tools > Inspector* erreichbar. Wie Sie den Menüeinträgen entnehmen können, existiert eine ganze Reihe von Inspektoren. In ihnen sind die Eigenschaften der Komponenten in Gruppen zusammengefasst. Erreichbar sind alle Inspektoren über ein einziges Fenster, das Reiter für die Bereiche *Attributes*, *Connections*, *Size* und *Identity* enthält. Abhängig von der selektierten Komponente können noch weitere Reiter auftauchen.

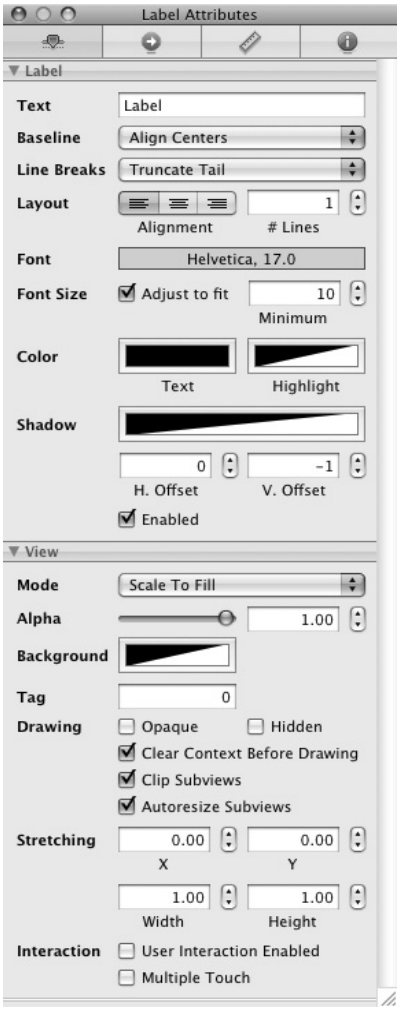


Bild 4.10: Der Attributes-Inspektor

Attributes

Im Reiter *Attributes* lassen sich spezifische Eigenschaften der ausgewählten Komponente bearbeiten. Bei einem Label sind das beispielsweise der darzustellende Text, die Textgröße, Schriftart und Farbe. Der Inhalt des Reiters sieht also für jede Komponente anders aus.

Size

Der *Size*-Inspektor ist zuständig für Größe, Ausrichtung und Vergrößerungsverhalten.

Identity

Beim Hinzufügen von Komponenten aus der Library werden stets zunächst die in der UIKit-Bibliothek enthaltenen Basisklassen wie `UIButton`, `UIViewController` usw. instanziiert. Oft werden Sie nun aber den Fall haben, dass Sie Instanzen Ihrer selbst geschriebenen Klassen, zum Beispiel eines eigenen View Controllers, erzeugen möchten. Hier hilft der Reiter *Identity* bzw. der Identity-Inspektor. Er erlaubt die Auswahl einer Klasse für eine Komponente. Eigene Klassen aus dem aktuellen Projekt tauchen hier automatisch auf. Der Interface Builder kennt also alle Ihre Xcode-Klassen und -Ressourcen. Im weiteren Verlauf werden wir praktische Beispiele hierfür sehen.

Connections

Das Gestalten der Views im Interface Builder macht Spaß und funktioniert ausgesprochen gut. Nun müssen die schicken Oberflächen irgendwie mit dem Anwendungscode verbunden werden. Genau das ist die Aufgabe des Reiters *Connections*.

Einerseits sollen bestimmte Komponenten, wie etwa Textfelder, mit Daten aus dem Model gefüllt oder Anwendereingaben aus diesen Controls ausgelesen werden. Es wird also eine Referenz im Code auf das UI-Element im Nib-File benötigt. Zum anderen sollen nach Betätigen von Buttons Aktionen ausgeführt, also Methoden im Code aufgerufen werden.

Die folgende Abbildung verdeutlicht den Zusammenhang:

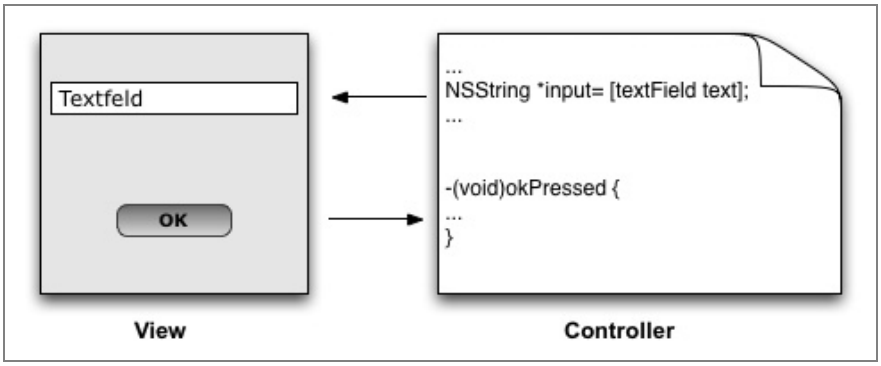


Bild 4.11: Zusammenspiel zwischen View und Controller

Für beide Fälle ist gesorgt, die passenden Konstrukte heißen *Actions* und *Outlets*. Dabei handelt es sich um speziell gekennzeichnete Methoden (Actions) bzw. Instanzvariablen (Outlets) im Controller des Views, der die zu verbindenden UI-Komponenten enthält.

Damit der Controller mit den präparierten Methoden und Variablen im Interface Builder mit dem View zusammengeführt werden kann, wird eine Instanz seiner Klasse im Interface Builder benötigt. Ein generischer View Controller kann aus der Library per Drag & Drop dem Document Window zugefügt werden. Die Klasse dieses Controllers wird anschließend im Identity-Inspektor auf die konkrete eigene Klasse gesetzt. Durch

diesen kleinen Kunstgriff wird der in Xcode durch Header und Implementation definierte View Controller im Interface Builder instanziiert.

Betrachten wir zunächst den Fall einer aufzurufenden Aktion näher. Die entsprechende Methode im Header der Controller-Klasse wird durch das Schlüsselwort `IBAction` anstelle des Rückgabewerts gekennzeichnet:

```
-(IBAction)okPressed;
```

Wird in der Implementierung der Methode eine Referenz auf den Sender (also etwa den Button) benötigt, kann folgende Variante benutzt werden:

```
-(IBAction)okPressed:(id)sender;
```

Da offen gelassen werden soll, wer die Methode aufrufen kann (wer also Sender des Events ist), wird für den übergebenen Parameter der generische Typ `id` gewählt.

Eine dritte Variante enthält zusätzliche Informationen über das aufgetretene Event:

```
-(IBAction)okPressed:(id)sender forEvent:(UIEvent*)event;
```

In der Implementierung der Methode steht ebenfalls `IBAction` anstelle von `void`.

Durch die Kennzeichnung `IBAction` ist für Interface Builder erkennbar, dass diese Methode von der GUI aus aufgerufen werden kann. Im Reiter *Identity* ist bei ausgewähltem *ViewController* im *Document Window* die gekennzeichnete Methode unter *Action* aufgeführt:

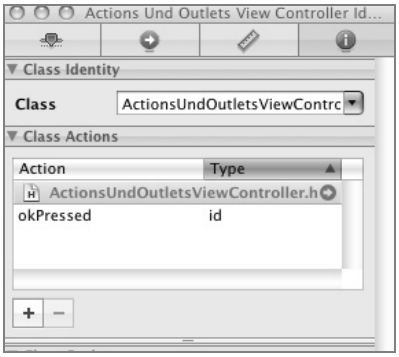


Bild 4.12: Action im Identity Inspector

Die Verbindung zwischen Button und Aktion wird wie angekündigt im Reiter *Connections* des Interface Builder erstellt. Auch hier sind die Actions und Outlets aufgelistet. Neben dem Namen der Action bzw. des Outlets findet sich ein kleiner Kreis. Durch Drag & Drop von diesem Kreis zur Komponente lässt sich die Verbindung ziehen:

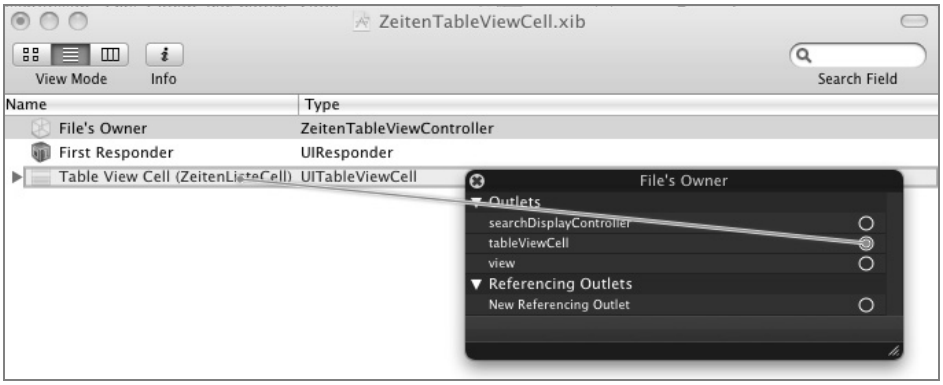


Bild 4.13: Connection erstellen

Das Vorgehen für die Outlets ist ganz ähnlich. Die Instanzvariable wird wie gehabt im Header-File deklariert. Zwecks Interface-Builder-Kennzeichnung bekommt sie den Vermerk `IBOutlet` vorangestellt:

```
IBOutlet UITextField *textField;
```

Interface Builder scannt die Klassen im Projekt und erkennt, dass es sich um eine Variable handelt, die mit der UI verbunden werden kann. Das Outlet der Controller-Klasse wird sowohl im Identity Inspector als auch im Connections Inspector angezeigt. Im *Connections*-Reiter kann wiederum aus dem kleinen Kreis heraus eine Verbindungslinie auf das Textfeld im View gezogen werden.

4.4 Instruments

Instruments ist ein Tracing- und Profiling-Tool. Mit dem Werkzeug lässt sich das Laufzeitverhalten von Programmen unter verschiedenen Aspekten wie zum Beispiel Anzahl und Speicherverbrauch der erzeugten Objekte, Speicherlecks, CPU-Aktivität oder Festplattenzugriffe näher unter die Lupe nehmen. Die während des *Instruments*-Laufs gesammelten Daten lassen sich für eine spätere Auswertung speichern.

Der Start des Tools erfolgt beispielsweise aus dem Xcode-Menü über die Option *Run > Run with Performance Tools > Leaks.*

Es präsentiert sich folgendes Fenster:

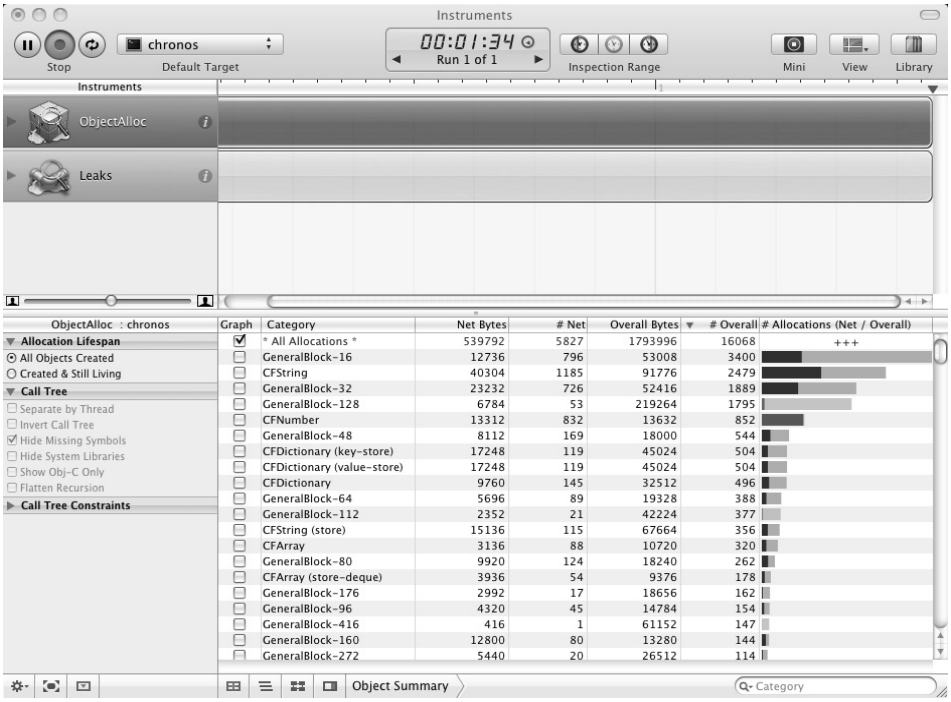


Bild 4.14: Das Instruments-Werkzeug

Im linken oberen View des Fensters lassen sich die momentan verwendeten Untersuchungsoptionen (die eigentlichen »Instrumente«) erkennen. Im Screenshot sind *ObjectAlloc* und *Leaks* zu sehen. Rechts daneben wird das Auftreten bestimmter Ereignisse während des Programmlaufs grafisch dargestellt. Durch Klick auf eines der Instrumente wird der untere Bereich des Fensters aktualisiert. Im linken unteren View lassen sich die Instrumente konfigurieren. Im verbleibenden Bereich rechts unten werden Details zu den aufgetretenen Ereignissen in tabellarischer Form aufgelistet.

In der Werkzeugleiste von *Instruments* findet sich der Button zum Öffnen der Library.

Bild 4.15: Instrumentenauswahl

Per Drag & Drop können daraus weitere Instrumente in den linken oberen View gezogen werden. Ein Klick auf den Button *Stop* in der Toolbar beendet die aktuelle Tracing-Sitzung (und die untersuchte Anwendung im Simulator oder auf dem Gerät). Schließt man das *Instruments*-Fenster, hat man die Möglichkeit, die erfassten Daten zu speichern.

Ein typisches Anwendungsbeispiel für Instruments, die Suche nach Speicherlecks, ist im Kapitel »Memory Management« enthalten.

5 Debugging

Die Entwicklungsumgebung Xcode bietet eine leistungsfähige grafische Benutzeroberfläche für den enthaltenen GNU Debugger (GDB). »Extreme Debugger«, die ihre Kommandos lieber von der Konsole aus absetzen, können das natürlich trotzdem tun.

Eine der häufigsten Fehlerquellen bei der iPhone-Entwicklung sind Speicherprobleme. Das Aufspüren von Leaks und Overreleases wird deshalb in einem separaten Kapitel, »Memory Management«, behandelt.

5.1 Konsolenausgaben

Die einfachste, wenn auch eingeschränkte Möglichkeit, ein Programm zu debuggen, sind Ausgaben auf der Konsole. Eigentlich sollte man auf diese Möglichkeit am Anfang ganz bewusst verzichten, um sich an den Umgang mit dem Debugger zu gewöhnen. Andererseits gibt es durchaus Situationen, in denen die Auflistung aller Ausgaben in der Konsole hilfreicher ist als die Momentaufnahme des Debuggers.

Konsolenausgaben werden mit der Klasse `NSLog` aus dem Foundation-Framework durchgeführt:

```
NSLog(@"Current language: %@", currentLanguage);
```

Beachten Sie an dieser Stelle bitte das `@`-Zeichen vor dem auszugebenden String. Strings in Objective-C werden grundsätzlich damit gekennzeichnet. Mit dem Ausdruck `%@` am Ende des Strings wird ein variabler Wert (ebenfalls vom Typ String) in den String eingebunden. Er folgt nach dem Komma. Beim Einstieg in Objective-C wird das vergessene `@`-Zeichen Ihr steter Begleiter und treuer Freund ...

Leider handelt es sich bei `NSLog` nicht um ein komplettes Logging-Framework mit Log-Leveln, tollen Formatierungsmöglichkeiten, Wahl des Ausgabeziels und all den anderen Annehmlichkeiten.

Reine Debug-Ausgaben können folglich bei der Auslieferung nicht durch Hochsetzen eines Log-Levels herausgefiltert werden und müssen notfalls von Hand entfernt oder zumindest auskommentiert werden. Eine Suche im Internet zu diesem Thema fördert glücklicherweise einiges an Makros und regulären Ausdrücken zutage, die dem Entwickler die Arbeit erleichtern.

Um typische Tracing-Informationen wie Klassenname, Methodenname usw. mit `NSLog` auszuschreiben, kann die Variable `__PRETTY_FUNCTION__` sehr nützlich sein:

```
NSLog(@"%s", __PRETTY_FUNCTION__);
```

Der Aufruf, ausgeführt in der Methode `applicationDidFinishLaunching:` der Klasse `chronosAppDelegate`, führt zu folgender Ausgabe:

```
2010-12-11 13:38:30.002 chronos[15684:20b] -[chronosAppDelegate
   applicationDidFinishLaunching:]
```

Richtig hilfreich werden `NSLog`-Ausgaben, wenn man die `description`-Methode von Objekten überschreibt. `description` ist eine Methode, die alle Objekte von `NSObject` erben und die im Standardfall den Typ und die Speicheradresse des Objekts ausgibt.

Die Anweisung `NSLog(@"%@", kunde)` für ein Objekt vom Typ `kunde` führt ohne Überschreiben der Methode zu einer Ausgabe dieser Art:

```
<Kunde: 0x103350>
```

Die Informationen sind natürlich wenig brauchbar. Spannender wäre es, die Werte der Instanzvariablen zu sehen.

Dazu wird die `description`-Methode, etwa für ein `Kunde`-Objekt, wie folgt überschrieben:

```
-(NSString *)description
{
   return [NSString stringWithFormat:@"Kunde [name: %@, bemerkung:
%@]",name,
     bemerkung];
}
```

Man erhält dadurch mit dem gleichen Aufruf eine deutlich aussagekräftigere Meldung:

```
Kunde [name: Müller AG, bemerkung: Schwer erreichbar]
```

Um später unnötige Verwirrungen durch das Beispiel zu vermeiden, sei an dieser Stelle darauf hingewiesen, dass bei Objekten vom Typ `NSManagedObject` (wie sie im Buch verwendet werden) die `description`-Methode nicht überschrieben werden sollte. Nähere Infomationen zu dem Thema finden Sie in der Klassendokumentation von `NSManagedObject`.

5.2 Debugger

Ein Debugger ist ein Werkzeug, mit dem der Ablauf (die monentan ausgeführte Anweisung) und der Zustand (die Werte der Variablen) der entwickelten App kontrolliert und sogar manipuliert werden können. Mit dem Debugger sehen Sie also quasi »unter die Motorhaube« Ihrer App. Damit der kompilierte Code die für das Debugging benötigten Debug-Informationen enthält, muss die Anwendung in der Debug-Konfiguration gestartet werden. Das ist im Normalfall nach dem Erzeugen eines neuen Projekts ohnehin gegeben und kann in der Overview-Auswahlliste in der Toolbar kontrolliert werden:

Bild 5.1: Auswahl der Konfiguration

Ein Start der Anwendung mit *Build and Go* startet die App während der Entwicklung immer im Debug-Modus. Der Entwickler ist also jederzeit in der Lage, nähere Informationen über den Zustand seiner Anwendung einzuholen. Er muss dazu nur einen Breakpoint setzen. Dabei handelt es sich um eine Kennzeichnung im Programmcode, an der der Debugger das Programm unterbricht und so die Inspektion der Variablen erlaubt.

Das Setzen des Breakpoints geschieht durch Klicken auf den grauen, breiten Rand (Gutter) links neben dem Text im Editor. Der aktive Breakpoint wird durch einen blauen Pfeil symbolisiert. Ein erneuter Klick darauf lässt den Pfeil erblassen. Der Breakpoint ist dann deaktiviert und bewirkt keinen Stopp des Debuggers mehr.

Zum Löschen des Breakpoints kann dieser einfach per Drag & Drop nach rechts auf das Editor-Fenster gezogen werden. Er löst sich dann, nett visualisiert, in einem Rauchwölkchen auf. Weitere hilfreiche Funktionen finden sich im Kontextmenü, das bei Rechtsklick auf den grauen Rand geöffnet wird.

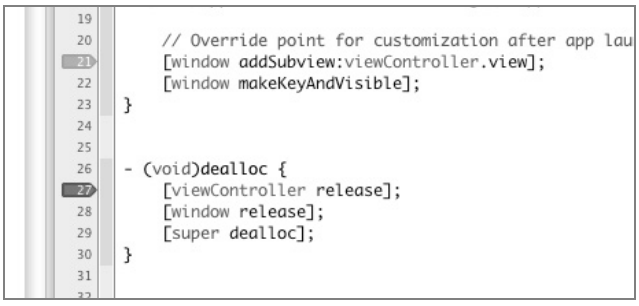


Bild 5.2: Inaktive und aktive Breakpoints

Wird der Breakpoint erreicht, hält der Debugger die Ausführung des Programms erwartungsgemäß an. In der Xcode-Statuszeile am unteren Fensterrand findet sich der Vermerk *GDB: Stopped at breakpoint...* Im Editor-Fenster kennzeichnet ein roter Pfeil die aktuelle Position des Debuggers im Code. Der rote Pfeil, und damit die aktuelle Position, kann per Drag & Drop innerhalb der Methode verschoben werden. Schiebt man den Pfeil nach oben, wird beim weiteren Durchlauf der darauf folgende Code erneut ausgeführt. Die bereits ausgeführten Statements werden also nicht etwa wie bei einem »Rückspulen« rückgängig gemacht.

Am oberen Rand des Editor-Fensters finden sich Buttons für die gängigen Debugger-Aktionen wie *Continue*, *Step over*, *Step into* und *Step out*:

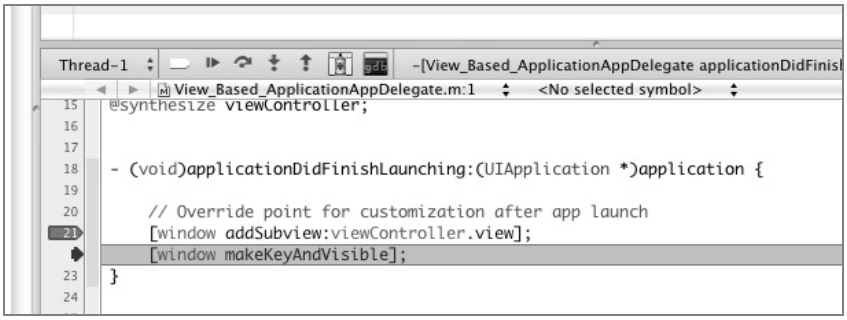


Bild 5.3: Der Debugger in Aktion

Continue lässt das Programm bis zum nächsten Breakpoint weiterlaufen.

Step over arbeitet den aktuellen Schritt ab und springt zum nächsten Schritt. Handelt es sich beim abzuarbeitenden Schritt um einen Methodenaufruf, hält der Degugger nicht in der Methode an.

Step into springt ebenfalls zum nächsten abzuarbeitenden Schritt, springt allerdings in Methoden hinein.

Step out verlässt die aktuelle Methode und springt zur nächsten Anweisung des aufrufenden Codes.

Weiterhin können in der Leiste ein separates Debugger-Fenster oder die Konsole geöffnet sowie der Stacktrace bis zu aktuellen Position untersucht werden.

Hält man während der Debug-Sitzung den Mauszeiger über eine Variable im Quellcode, wird sie rot umrandet und eine Auswahlliste wird eingeblendet. Die Auswahlliste erlaubt das Inspizieren der Werte eines Objekts. Bei verschachtelten Objekten kann die Auswahlliste weiter aufgeklappt werden, um an die »inneren Werte« zu gelangen.

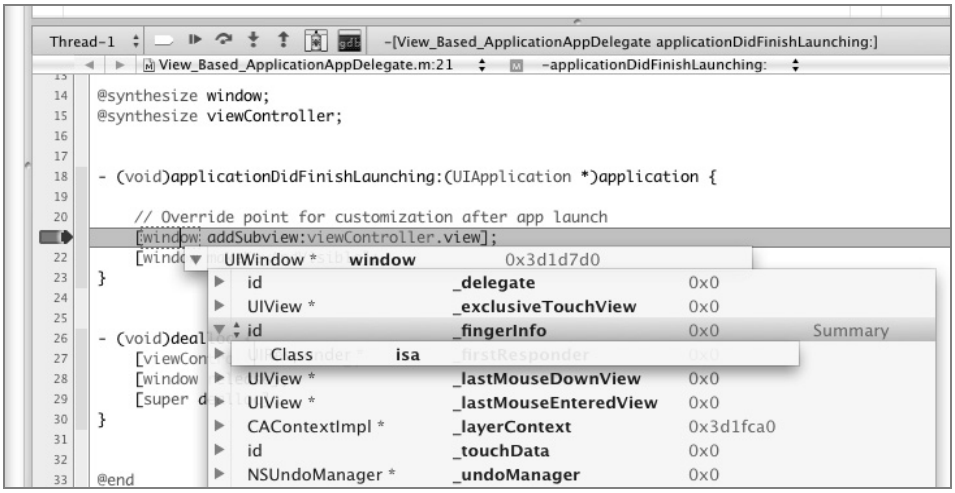


Bild 5.4: Inspizieren eines Objekts

Die Variablen können nicht nur betrachtet, sondern sogar während der Debug-Session verändert werden. Dazu klickt man einfach in die Auswahlliste auf den zu verändernden Wert und gibt ihn im erscheinenden Textfeld in der gewünschten Variante ein.

Alternativ können die Werte der Variablen auch im separaten Debugger-Fenster untersucht werden. Die Option *Run > Debugger* aus dem Menü öffnet das Fenster, das einen guten Überblick über alle Variablen im aktuellen Scope bietet (siehe folgendes Bild).

Ein aktivierter Breakpoint führt im Normalfall bei jedem Durchlauf zum Stopp der Codeausführung. Oft hat man aber den Fall, dass nur unter bestimmten Bedingungen gestoppt werden soll, etwa wenn eine Variable einen bestimmten Wert hat. Gängiger Workaround, um das Verhalten zu erzielen, ist ein `if`-Statement, das die entsprechende Bedingung prüft, kombiniert mit einem Breakpoint innerhalb des `if`-Blocks. Da der Code nur bei erfüllter Bedingung durchlaufen wird, greift der Breakpoint auch nur in diesem Fall.

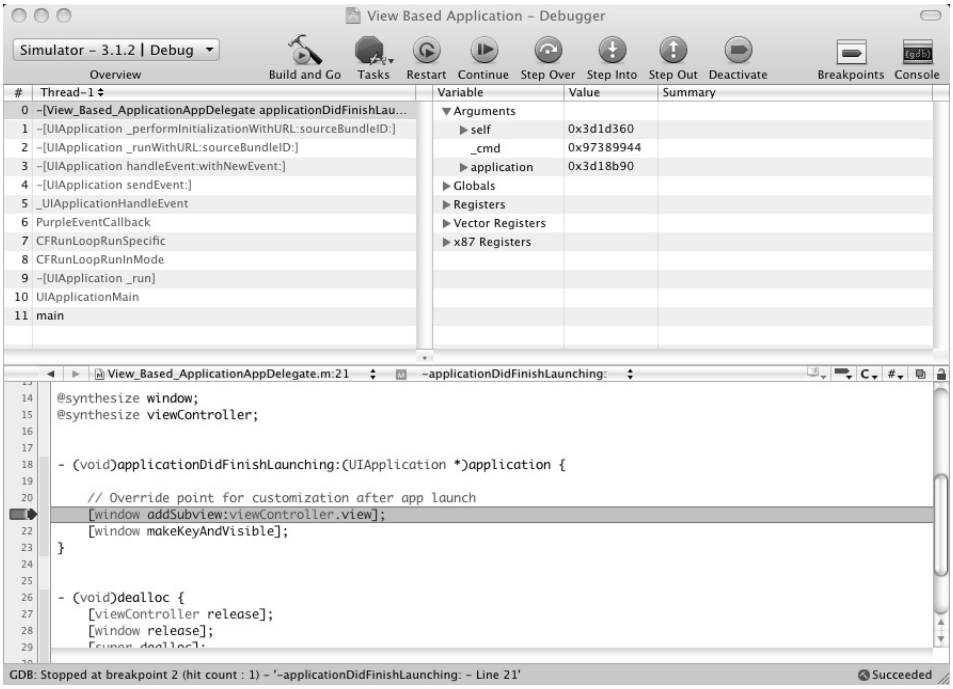


Bild 5.5: Das Debugger-Fenster

Eleganter kann das Verhalten mit einem konditionalen Breakpoint erreicht werden. Man öffnet dazu im Breakpoint-Kontextmenü über den Eintrag *Edit Breakpoint* das Breakpoint-Fenster:

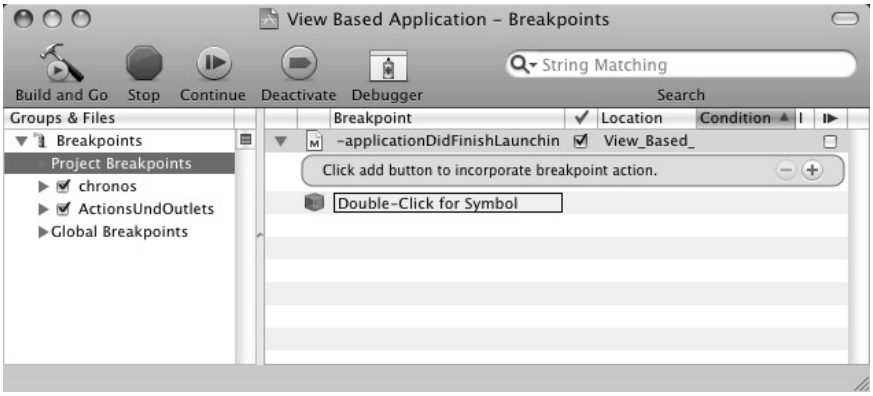


Bild 5.6: Das Fenster *Edit Breakpoint*

Hier werden alle im Projekt vorhandenen Breakpoints aufgelistet. In der Spalte *Condition* können beliebige Ausdrücke angegeben werden, die einen Boole'schen Wert als Ergebnis haben. Nur wenn die Bedingung erfüllt ist, greift der Breakpoint.

Eine ebenfalls in der Praxis oft auftretende Anforderung ist die, dass ein Breakpoint in einer Schleife erst beim x-ten Durchlauf zum Anhalten der Ausführung führen soll. Abhilfe schafft hier die Option *Ignore Count* direkt hinter *Condition*. Setzt man den Wert beispielsweise auf vier, hält der Debugger erst beim fünften Durchlauf an.

Wer schon immer mal den Wunsch hatte, sich einen Breakpoint vorlesen (!) zu lassen, kann das über verknüpfte Aktionen tun. Mit dem Pluszeichen unterhalb des Breakpoints lassen sich aber auch sinnvollere Tätigkeiten, wie zum Beispiel das Schreiben von Log-Meldungen, verknüpfen.

5.3 Remote Debugging

Zugegeben, dieser Abschnitt hätte entfallen können. Debugging funktioniert bei angeschlossenem iPhone (fast) genau wie mit dem Simulator. Der Debugger GDB verbindet sich remote mit dem Prozess auf dem externen Gerät und erlaubt so eine ausgesprochen komfortable Fehlersuche.

Log-Meldungen und andere Ausgaben werden wie gehabt in der Konsole angezeigt.

Ist das Gerät nicht angeschlossen, so können aufgetretene Crash Logs und Log-Meldungen im Organizer nach dem Verbinden mit dem USB-Kabel angezeigt werden. Die Daten werden vom Gerät übertragen.

Auf dem Gerät sind einige wenige, fortgeschrittene Debugging-Hilfsmittel wie *GuardMalloc, DTrace* und *NSZombie* nicht nutzbar. Sollte man diese benötigen, muss auf den Simulator zurückgegriffen werden.

6 Memory Management

Willkommen in einem der wichtigsten Kapitel in diesem Buch. Kommt Ihnen Folgendes bekannt vor? Sie haben sich aus dem App Store eine neue Anwendung geladen, vielleicht eine dieser eher begrenzt hilfreichen Taschenlampen-Anwendungen. Sie spielen ein wenig damit herum und plötzlich finden Sie sich auf dem Home-Screen wieder. Kein Warnhinweis, keine Fehlermeldung, einfach rausgeworfen. Was ist passiert? Mit einiger Wahrscheinlichkeit sind Sie Opfer eines Speicherproblems geworden.

In jeder objektorientierten Programmiersprache muss der Platz, den die Objekte belegen, irgendwann wieder freigegeben werden. Ansonsten läuft der Speicher voll, und das Programm wird beendet. Das Freigeben kann entweder automatisch durch das System erledigt werden, oder aber die Aufgabe bleibt am Entwickler hängen. Apple hat mit Objective-C 2.0 die *Garbage Collection*, also das automatische Freigeben des Speichers, eingeführt. Wenn Sie Cocoa-Anwendungen für den Mac entwickeln, brauchen Sie sich deshalb, wie auch in Java oder C#, über dieses Thema keine Gedanken mehr zu machen.

Anders sieht die Welt leider (noch) für den iPhone-Entwickler aus. *Garbage Collection* kostet Performance, und der Prozessor des mobilen Geräts soll mit dem Feature nicht belastet werden. Stattdessen wird vorerst der Programmierer mit dem eher rückständigen und fehlerträchtigen Überwachen der Objektanzahl betraut. Es bleibt zu hoffen, dass diese Entscheidung mit steigender Prozessorleistung im Hause Apple noch mal überdacht wird. Aber alles Jammern hilft nicht. So wie die Dinge im Moment stehen, müssen wir uns dringend mit dem Thema beschäftigen.

6.1 Zwei Regeln

Grundlage des Memory-Management-Modells sind zwei Regeln, die definieren, um welche Objekte sich der Entwickler kümmern muss:

1) Eigene Objekte müssen mit `release` oder `autorelease` freigegeben werden.

2) Der Speicher von fremden Objekten wird auf keinen Fall freigegeben.

Da stellt sich natürlich die Frage, welche Objekte einem denn gehören. Auch hierfür wurden klare Regeln definiert:

1) Das Objekt wurde mit einer Methode erzeugt, die mit `alloc` oder `new` beginnt (zum Beispiel `newObject`).

2) Das Objekt wurde mit einer Methode von einem anderen Objekt kopiert, die `copy` enthält (zum Beispiel `mutableCopy`).

3) Die (Speicher-)Verantwortung für ein Objekt wurde durch Senden der `retain`-Nachricht übernommen.

Diese Regeln sind keineswegs als Empfehlung zu betrachten. Sie müssen eingehalten werden, denn das Memory-Management-Modell funktioniert nur, wenn sich alle Entwickler an die Spielregeln halten.

6.2 Retain, release & dealloc

Nachdem nun bekannt ist, für welche Objekte wir verantwortlich sind, muss geklärt werden, wie der Speicher freizugeben ist. Dazu muss man wissen, dass für jedes Objekt der Anwendung ein `retainCount`, ein Zähler der Objektbesitzer, existiert.

Wenn ein Objekt mit `alloc`, `new` oder `copy` erzeugt wird, dann wird der Zähler des Objekts zunächst auf eins gesetzt.

Wird für das Objekt die `release`-Methode aufgerufen, wird der Zähler um eins gesenkt. `release` bedeutet also, dass das Objekt »von uns aus« gelöscht werden kann. Aber nur von uns aus, weil ja eventuell noch andere Besitzer existieren (die sich durch `retain` eingekauft haben).

Wird nach der Erzeugung die `retain`-Methode für ein Objekt aufgerufen, wird der Zähler um eins erhöht. `retain` ruft man auf, wenn man die Verantwortung für ein Objekt übernehmen möchte, das man durch einen Methodenaufruf erhalten hat. Das Objekt wurde an anderer Stelle (in einer anderen Klasse) erzeugt und gehört uns nicht. Durch `retain` signalisieren wir, dass das Objekt von uns benötigt wird und nicht gelöscht werden darf, bevor wir zustimmen. Diese Situation tritt immer dann auf, wenn man das erhaltene Objekt in einer Instanzvariablen über die aktuelle Methode hinaus halten möchte. Innerhalb der Methode, in der man das Objekt erhalten hat, bleibt es ohnehin gültig.

Der aktuelle Zählerstand kann über die `retainCount`-Methode ausgegeben werden:

```
MeinObjekt *meinObjekt = [[MeinObjekt alloc]init];
NSLog(@"retainCount: %d," [meinObjekt retainCount]);
```

Den Nutzen der `retainCount`-Ausgabe sollte man nicht überschätzen. Auch Framework-Methoden senden `retain` an übergebene Objekte, sodass das Ergebnis schwer vorhersagbar ist und bei der Fehlersuche oft nicht wirklich weiterhilft.

Erreicht der `retainCount` für ein Objekt den Wert 0, dann wird automatisch die `dealloc`-Methode für das Objekt aufgerufen. `dealloc` wird immer nur durch das System und nie durch den Entwickler selbst aufgerufen. Die Aufgabe der `dealloc`-Methode ist es, den Speicher des Objekts und alle weiteren Ressourcen freizugeben. Zu den weiteren Ressourcen gehören insbesondere die Instanzvariablen.

Sie sind vom Entwickler durch Überschreiben der `dealloc`-Methode freizugeben:

```
- (void)dealloc {
[adresse release];
[name release];
[super dealloc];
}
```

Der Aufruf von `dealloc` in der Superklasse muss am Ende der Methode erfolgen.

Schauen wir uns ein Beispiel aus der Praxis an:

```
TextViewEditController *controller = [[TextViewEditController alloc]
          initWithNibName:@"TextViewEditView" bundle:nil];
controller.editedObject = leistung;
controller.editedFieldKey = @"bemerkung";
controller.editedFieldName = @"Bemerkung";
[self.navigationController pushViewController:controller animated:YES];
[controller release];
```

Wir (unsere Klasse) erzeugen im Beispiel einen `ViewController` mit `alloc` und haben somit die Pflicht, den Speicher wieder freizugeben. Der `ViewController` wird ein wenig konfiguriert und anschließend mit der Methode `pushViewController:animated:` einem `navigationController`-Objekt übergeben. Anschließend benötigen wir das `controller`-Objekt nicht weiter und geben unseren Segen zum Löschen durch den Aufruf der `release`-Methode. Der Controller wird dadurch aber nicht sofort gelöscht. Lediglich der `retainCount` wird verringert. In obigem Fall ist anzunehmen, dass der `navigationController` in seiner Methode `pushViewController:animated:retain` aufruft, um sein Interesse an dem Objekt zu bekunden. Der `retainCount` steht also vermutlich nach unserem `release` immer noch auf eins und das Objekt existiert weiter. Irgendwann braucht auch der `NavigationController` den `ViewController` nicht mehr und ruft `release` auf. Dazu hatte er sich gemäß den Regeln verpflichtet, weil er zuvor `retain` aufgerufen hatte. Nun steht der `retainCount` des Objekts auf null und die `dealloc`-Methode wird aufgerufen.

Sehen wir uns ein zweites Beispiel an. Im folgenden Code wird ein Array mit `NSNumber`-Objekten gefüllt:

```
NSMutableArray *aArray = [[NSMutableArray alloc]init];
NSUInteger i;
// ...
for (i = 0; i < 5; i++) {
   NSNumber *aNumber = [NSNumber numberWithInteger:i];
   [aArray addObject:aNumber];
}
```

Die `Number`-Objekte werden über eine Komfort-Methode **ohne** `alloc`, `new` oder `copy` erzeugt. Somit dürfen sie keine `release`-Nachricht gesendet bekommen.

Anders sieht es aus, wenn das `Number`-Objekt mit `alloc` erzeugt wird:

```
NSMutableArray *aArray = [[NSMutableArray alloc]init];
NSUInteger i;
// ...
for (i = 0; i < 5; i++) {
   NSNumber *aNumber = [[NSNumber alloc] initWithInteger: i];
   [aArray addObject:aNumber];
   [aNumber release];
}
```

In diesem Beispiel wird die Nummer mit `alloc` erzeugt und dem Array hinzugefügt. Folglich ist die Klasse, in der sich der Code befindet, zuständig für das Freigeben des Speichers, was in der nächsten Zeile auch passiert. Trotz des `release` ist das Nummern-Objekt im Array aber weiterhin vorhanden. Die Array-Klasse ruft in der `addObject:`-Methode `retain` auf und erhöht den Zähler somit um eins.

6.3 AutoreleasePool

Es gibt Fälle, in denen das bisher beschriebene Vorgehen nicht angewendet werden kann. Ein typisches Beispiel sind Methoden, die ein Objekt erzeugen und zurückgeben sollen:

```
-(Projekt*) createProjekt {
   Projekt *projekt = [[Projekt alloc]init];
   [projekt release]; // Fehler!
   return projekt;
}
```

Wenn diese Methode von einer anderen Klasse aufgerufen wird, wer ist dann zuständig für das Freigeben des Speichers? Gemäß den Regeln der Besitzer, also die Klasse, in der die Methode steht. Dummerweise kann das Objekt vor der Rückgabe nicht freigegeben werden. Die `dealloc`-Methode würde sofort zuschlagen und das Objekt beseitigen. Gesucht ist also ein Weg, das Objekt freizugeben, nachdem die aufrufende Klasse es auch benutzen konnte.

Für diese und ähnliche Fälle existiert der `AutoreleasePool`. Es handelt sich um ein großes Sammelbecken für zu löschende Objekte. Der Pool wird beim Start der Anwendung angelegt. Bei Verwenden einer der Xcode-Projektvorlagen findet das Erzeugen in der `main`-Methode statt:

```
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int retVal = UIApplicationMain(argc, argv, nil, nil);
   [pool release];
   return retVal;
}
```

Im weiteren Programmverlauf können zu löschende Objekte dem Pool durch Aufruf der Methode `autorelease` hinzugefügt werden:

```
-(Projekt*) createProjekt {
   Projekt *projekt = [[Projekt alloc]init];
   [projekt autorelease];
   return projekt;
}
```

Für alle Objekte im Pool wird die `release`-Methode aufgerufen, wenn der Pool selber freigegeben wird. Nutzt man den `AutoreleasePool` in der `main`-Methode des Templates, so werden die darin enthaltenen Objekte auch erst bei Beenden des Programms freigegeben. Dadurch kann der Speicherverbrauch bei vielen `autorelease`-Objekten natürlich sehr stark anwachsen. Um dem Problem zu begegnen, kann der Entwickler weitere `AutoreleasePools` erstellen und sie früher löschen.

Kommt Ihnen das Ganze ein wenig kompliziert vor? Ist es auch. Durch die Verteilung der Objekte und ihrer Besitzer über den ganzen Programmcode erfordert es viel Übung und Disziplin, den Überblick zu behalten und kein `retain` oder `release` zu vergessen. Zum Glück gibt es ein paar Hilfsmittel, um herauszufinden, ob und wo Fehler durch die Speicherverwaltung entstehen.

Grundsätzlich sind zwei Arten von Problemen denkbar: Ein Objekt wurde overreleased, das heißt, der `retainCount` ist schon auf null, obwohl das Objekt noch benutzt werden soll. Oder es bestehen Lecks (Leaks) durch nicht freigegebene Objekte.

6.4 Overrelease

Auf ein Objekt, das bereits freigegeben wurde, kann nicht mehr zugegriffen werden. Versucht man dies trotzdem, hält die Anwendung an und man wird mit einer mehr oder weniger kryptischen Fehlermeldung abgemahnt. In der Konsole erscheint im günstigsten Fall eine Mitteilung wie »`FREED(id): message setAccessoryType: sent to freed object=0x4f4711`«. Hier erlaubt der Name der Methode den Rückschluss auf das betroffene Objekt. In anderen Situationen hält die Anwendung ganz ohne Meldung an. Versucht man sie dann mit dem Debugger-Befehl `Continue` weiterlaufen zu lassen, folgt »`GDB: Program received signal: EXC_BAD_ACCESS`«. Dummerweise gibt die Fehlermeldung dann nicht den kleinsten Hinweis auf das betroffene Objekt.

An dieser Stelle tun wir etwas sehr Unerwartetes: Wir rufen die Zombies zu Hilfe. Ja, Sie haben richtig gelesen. Der ungewöhnliche Name ist gar nicht so unpassend für das Hilfsmittel, wie es zunächst scheint. Da das gesuchte Objekt bereits gelöscht, also quasi tot ist, stehen keine Informationen für die Fehlermeldung zur Verfügung. Wenn wir es aber als »Untoten« vorhalten, dann können wir es fragen, von welchem Typ es vor der Freigabe war.

Zum Aktivieren des Features rechtsklicken Sie auf das Executable Ihres Projekts (unter *Executables* in *Groups & Files*) und öffnen im Kontextmenü via *Get Info* den Info-Dialog.

Im Reiter *Arguments* können im unteren Bereich Umgebungsvariablen gesetzt werden. Hier wird die Variable `NSZombieEnabled` hinzugefügt und der Wert auf `YES` gesetzt:

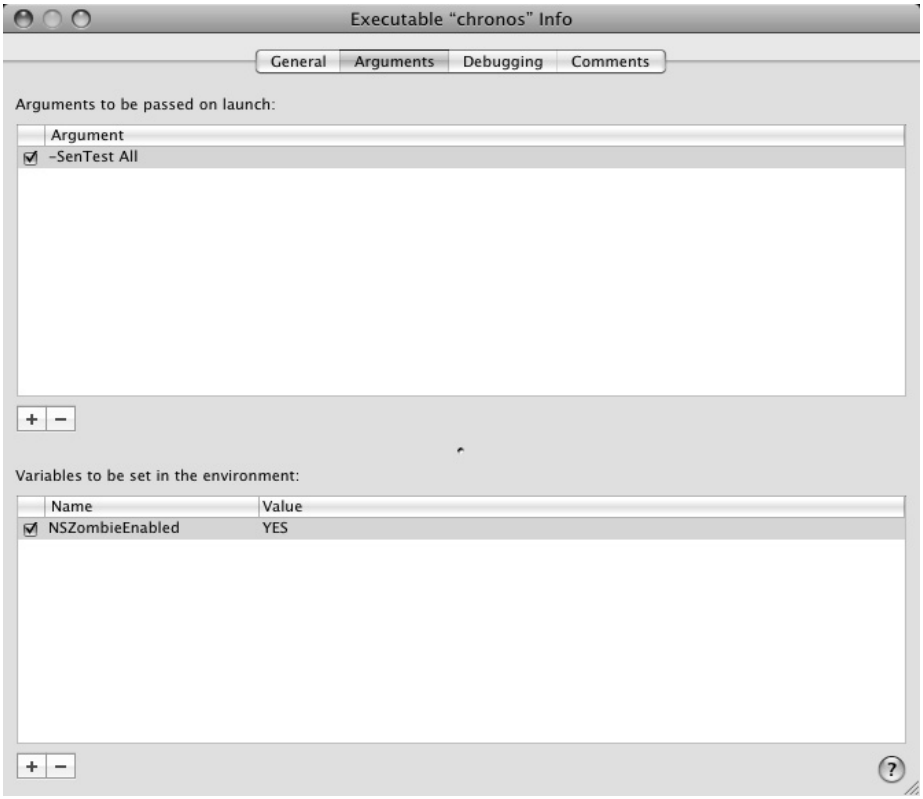


Bild 6.1: Die Umgebungsvariable `NSZombieEnabled`

Wenn man sich nun in der Konsole die Fehlermeldungen anschaut, ist zumindest der Typ des overreleasten Objekts erkennbar. Hier wurde eine Nachricht an eine bereits deallokierte Instanz eines Objekts vom Typ `TableViewCell` versendet:

Bild 6.2: Konsolenausgabe mit `NSZombieEnabled`

Nicht toll, aber ein Anfang, oder? Die Kenntnis des betroffenen Objekttyps vereinfacht die Fehlersuche schon deutlich.

Nach der Benutzung von `NSZombieEnabled` sollte die Variable durch die Checkbox im Info-Dialog deselektiert werden, weil zum Beispiel die Lecksuche mit Instruments dadurch gestört wird.

In neueren Versionen der im Kapitel »Entwicklungswerkzeuge« vorgestellten Anwendung Instruments ist die Suche mit *NSZombie* ebenfalls möglich. Dazu wird im Xcode-Menü die Option *Run > Run with Performance Tools > Object Allocations* ausgewählt. Nach dem Start wird Instruments mittels *Stop* gleich wieder angehalten und das kleine »i« auf der violetten *ObjectAlloc*-Schaltfläche links oben angeklickt. Im sich daraufhin öffnenden Dialog kann die Option *Enable NSZombie detection* selektiert werden:

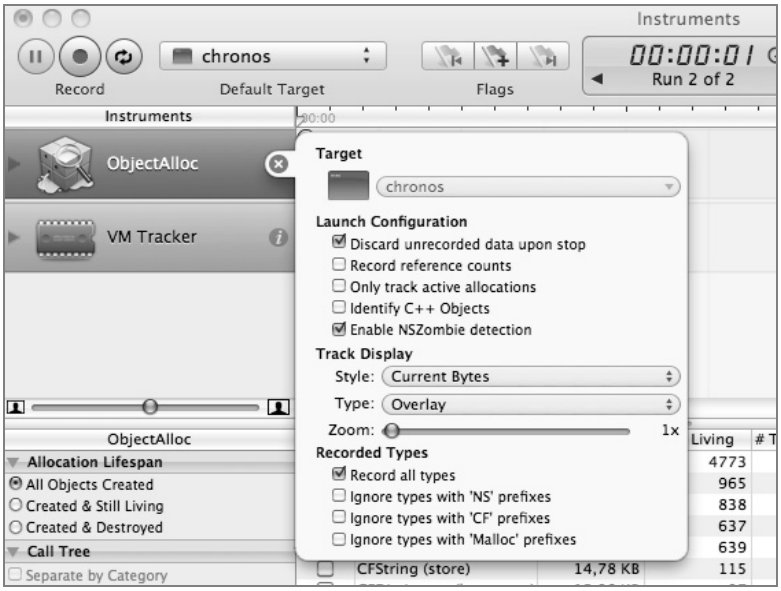


Bild 6.3: NSZombie-Detection in Instruments

Mit *Record* wird die Aufnahme wieder gestartet. Während des Instruments-Laufs wird die Anwendung ganz normal bedient, und alle zu untersuchenden Bereiche werden durchgeklickt. Wird ein overreleastes Objekt gefunden, meldet sich Instruments:

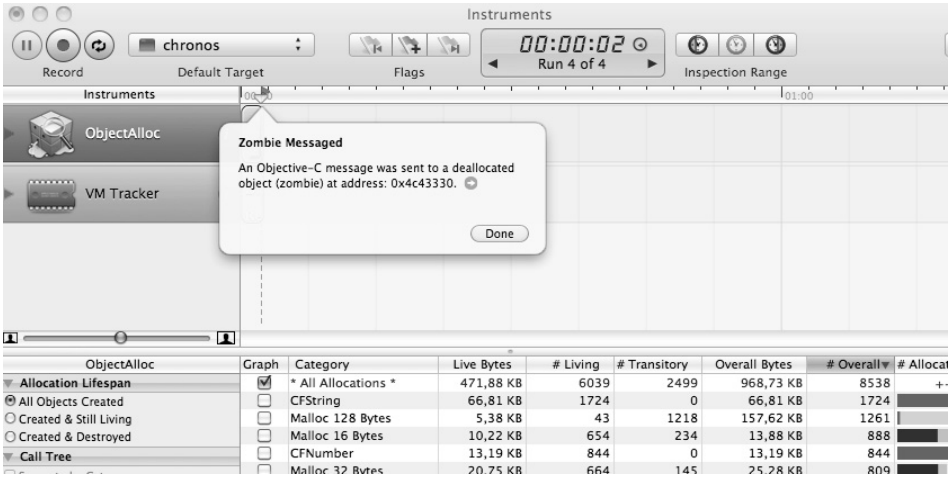


Bild 6.4: Instruments hat ein Zombie-Objekt entdeckt

Durch Klick auf den Pfeil in der Meldung und Untersuchen des Stacktrace kann dann nach der Ursache des Problems gesucht werden.

Der Stacktrace wird durch die Option *Extended Detail View* am unteren Ende auf der rechten Fensterseite eingeblendet. Er sollte mithilfe des Zahnrads auf der rechten Seite noch angepasst werden. Sehr sinnvoll ist die Option *File Icon*. Dadurch werden die eigenen Klassen mit einem kleinen Symbol gekennzeichnet, was das Auffinden ungemein erleichtert. Ein Doppelklick auf den Stacktrace führt im Code an die gesuchte Stelle.

6.5 Leaks

Im Fall nicht freigegebener Objekte treten die Probleme erst auf, wenn durch die angesammelten Objekte der Speicher knapp wird. Da das je nach Größe des Objekts und Häufigkeit des Auftretens eine Weile dauern kann, übersieht man Lecks während der Entwicklung sehr leicht. Erst ein Anwender, der die App längere Zeit bedient, bemerkt dann den Fehler.

Die Lösung für dieses Problem ist das regelmäßige, gezielte Suchen nach Lecks während der Entwicklung. Zum Aufspüren von Leaks wird ebenfalls Instruments benutzt. Die Lecksuche kann aus Xcode heraus mit *Run > Start with Performance Tools > Leaks* erfolgen.

Das untere der beiden Werkzeuge gibt Informationen über Speicherlecks preis. Die zugehörige Detaildarstellung sollte durch Anklicken der violetten *Leaks*-Schaltfläche aktiviert werden. Aufgetretene Lecks werden sowohl durch rote Peaks in der oberen Darstellung als auch durch Einträge in der Tabelle unten dargestellt.

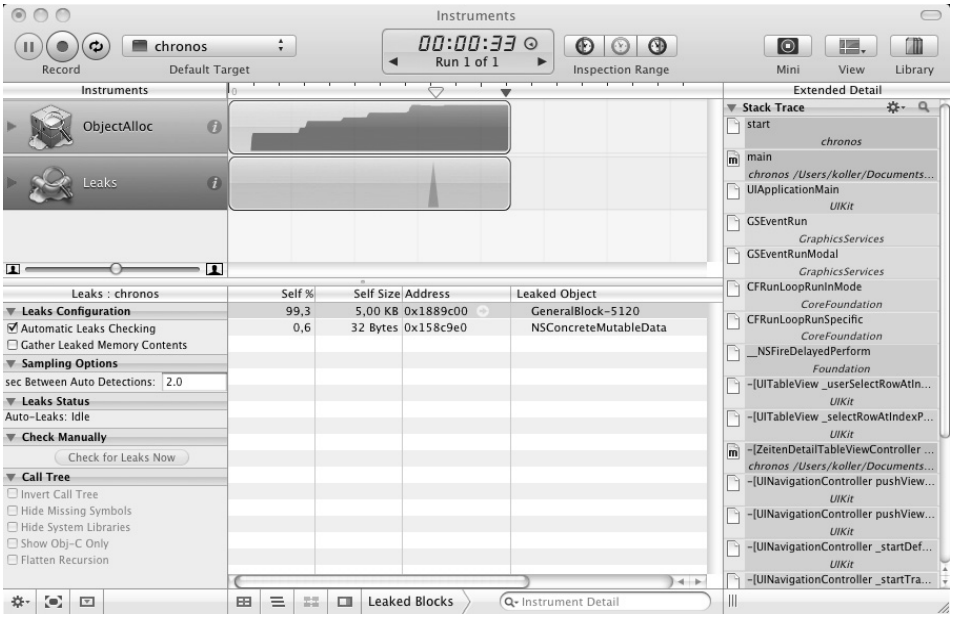


Bild 6.5: Ein Speicherleck in Instruments

Um möglichst zeitgleich zur Bedienung der Anwendung Informationen über Lecks zu erhalten, empfiehlt es sich, in den *Sampling Options* die Zeit zwischen Suchvorgängen (*sec Between Auto Detections*) zu reduzieren, beispielsweise auf zwei Sekunden.

Wurden ein oder gar mehrere Leaks gefunden, wird zunächst Instruments gestoppt. Anschließend wird ein Leak in der Tabelle markiert und im Stacktrace nach der Stelle gefahndet, an der das Leck auftritt.

Ab Mac OS 10.6 (Snow Leopard) gibt es in Xcode eine weitere hilfreiche Funktion, die beim Auspüren von Leaks (aber auch anderen Problemen) nützlich ist: die statische Codeanalyse. Der Quelltext wird dabei während des Build-Vorgangs einer Reihe von formalen Prüfungen unterzogen. Das Werkzeug wird mit *Build > Build and Analyze* gestartet und kommt zunächst ohne Ihre Hilfe aus.

Die folgende Abbildung zeigt das Ergebnis einer solchen Analysesitzung:

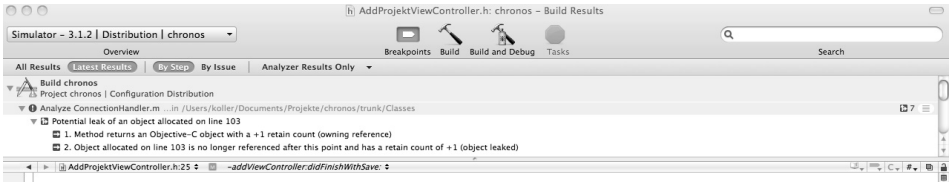


Bild 6.6: Statische Codeanalyse

Das Ergebnis der Untersuchung ist eher als Verdacht denn als Fakt zu interpretieren und anschließend von Ihnen zu überprüfen. Die Möglichkeiten der statischen Codeanalyse hängen stark von der Qualität der verwendeteten Algorithmen ab und sind durch die Durchführung zur Kompilierzeit begrenzt. Letztlich kennen Sie Ihren Code am besten und können entscheiden, ob hier wirklich ein Leak vorliegt oder der Analyzer an die Grenzen seiner Möglichkeiten gestoßen ist. Vor dem Ausliefern der fertigen Software sollte auf jeden Fall noch mal allen angezeigten Problemen nachgegangen werden.

Teil 2 – Das Grundgerüst der Zeiterfassung

Nachdem nun die Grundlagen besprochen sind, wird im folgenden Teil das Grundgerüst für die Zeiterfassung erstellt. Nach Konzeption, Anlegen des Projekts und Erstellen des Datenmodells stehen die verschiedenen View Controller im Vordergrund. Mit ihrer Hilfe werden das Hauptmenü, Listen-Views, Detail-Views und Bearbeiten-Views implementiert.

7 User Interface Design

Nachdem die Entwicklungswerkzeuge vorgestellt und die Besonderheiten der Sprache zumindest angesprochen sind, steht der Entwicklung Ihrer Killer-App nicht mehr viel im Weg. Wichtigster Baustein für ein erfolgreiches Gelingen ist bei der iPhone-Entwicklung – mehr noch als bei der Desktop- oder Webentwicklung – ein stimmiges Konzept. Das Konzept sollte beschreiben, welche Funktionalitäten in der Anwendung enthalten sind und wie diese umgesetzt werden.

Der Schwerpunkt dieses Kapitels liegt dabei auf der konzeptuellen Erarbeitung des User-Interface-Designs. Das Design der Anwendungsarchitektur, wie etwa die Verteilung von Verantwortlichkeiten an Klassen, wird weitgehend durch die in Cocoa realisierten und bereits besprochenen Design Patterns vorgegeben.

7.1 Das Gerät kennenlernen

Haben Sie schon einmal das Benutzerhandbuch für eine App gelesen? Wohl kaum.

Die meisten der mitgelieferten oder ladbaren Programme lassen sich intuitiv bedienen. Bei einer durchschnittlichen Nutzungszeit von 15 bis 20 Sekunden erwartet der Anwender, dass er die Programme ohne große Einarbeitung nutzen kann. Erfüllt eine Anwendung diese Erwartung nicht, wird sie sehr schnell wieder gelöscht.

Die intuitive Benutzung, die für den Benutzer eine Selbstverständlichkeit darstellt, ist für den Entwickler harte Arbeit und das Ergebnis eines erfolgreichen Designprozesses.

Als Neueinsteiger in die iPhone-Entwicklung sollten Sie sich zunächst sehr gründlich mit dem Gerät und den enthaltenen Anwendungen vertraut machen. Wie lassen sich mitgelieferte Programme wie Mail, Einstellungen oder iPod bedienen? Wie wird die fehlende Maus kompensiert? Welche UI-Controls erkennen Sie von Anwendung zu Anwendung wieder? Wie wird die Navigation realisiert? Welches Geheimnis steckt hinter der intuitiven Bedienung? Kann man sich vielleicht doch mit dem Gerät rasieren?

Nehmen Sie sich Zeit, um das Look & Feel einiger Anwendungen zu analysieren – die Investition lohnt sich!

7.2 Grundlegender Aufbau von Apps

iPhone-Apps lassen sich grob in drei Kategorien unterteilen. Gemeint sind dabei nicht etwa die Kategorien aus dem App Store, sondern eher eine technische Unterteilung:

- Utility (zum Beispiel *Aktien* oder *Wetter*)
- Produktivität (zum Beispiel *Mail* oder *Einstellungen*)
- Custom (Spiele)

Utility-Anwendungen sind kleine Hilfsmittel, deren Flipchart User Interface nur aus zwei Views besteht. Auf dem ersten View werden die eigentlichen Nutzinformationen dargestellt. Auf dem zweiten View, auf der Rückseite der Anwendung, kann der Benutzer Einstellungen tätigen.

Custom-Apps haben ein komplett selbst gestaltetes User Interface. Typische Vertreter sind aufwendige Spiele, die mithilfe spezieller Entwicklungsumgebungen erstellt werden und auf eigenen Engines ablaufen. Wer sich dafür interessiert, sollte sich zum Beispiel *Unity* [URL-UNITY] ansehen.

Wir konzentrieren uns hier auf die Kategorie *Produktivität*. Die Anwendungen dieses Typs machen den größten Gebrauch von der UIKit-Komponentenbibliothek.

Die folgende Abbildung zeigt exemplarisch den grundlegenden Aufbau vieler Anwendungen dieser Kategorie. Der Aufbau sollte Ihnen nach Untersuchung einiger Apps vertraut sein:

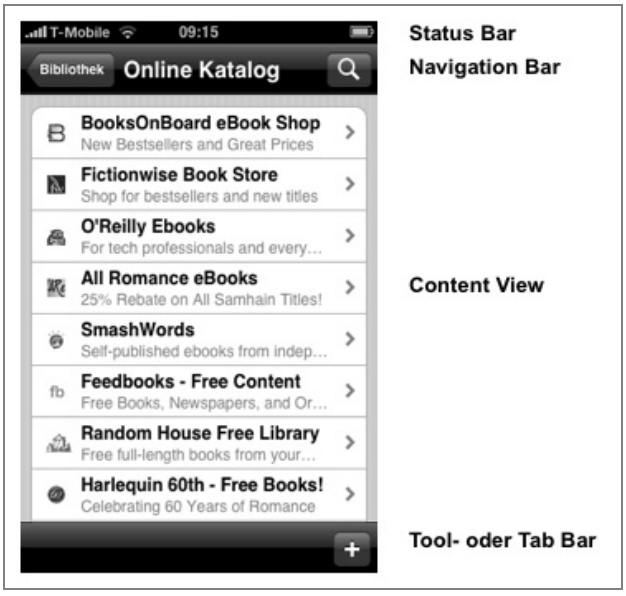


Bild 7.1: Aufbau einer Productivity-App

Natürlich gibt es reichlich Anwendungen, die beispielsweise ohne Tool- oder Tab-Bar auskommen oder die ein *Page Control*, etwa zur Darstellung von Bildern, verwenden. Es geht hier also nicht um eine scharfe Unterteilung, sondern um eine grobe Orientierung.

7.3 UI-Komponenten

Im Folgenden werden die wichtigsten Komponenten zur Strukturierung des User Interface kurz vorgestellt. Zur Erzeugung und Nutzung im Code finden sich im weiteren Verlauf des Buchs viele Beispiele.

7.3.1 Status Bar

Am oberen Rand des Displays befindet sich fast immer die *Status Bar*. Sie ist nicht Teil der Anwendung und enthält Informationen über Zellenstärke, Netzwerk, die aktuelle Uhrzeit sowie den Ladezustand des Akkus. Bei Bedarf kann die *Status Bar* mit folgendem Aufruf ausgeblendet werden:

```
[[UIApplication sharedApplication] setStatusBarHidden:YES];
```

Viele Spiele machen von dieser Möglichkeit Gebrauch, um den vollen Platz des Displays nutzen zu können. Für Anwendungen des Typs *Produktivität* empfiehlt es sich in den meisten Fällen nicht. Oder möchten Sie den Nutzer nötigen, Ihre App zu beenden, nur weil er wissen möchte, wie spät es ist?

Ein schönes Feature der *Status Bar* ist die Anzeige eines Aktivitätsindikators. Er sollte bei längeren Zugriffen auf das Netzwerk angezeigt werden. Anwendungen wie Mail oder Safari nutzen diese Option, die sich mit einer Zeile Code implementieren lässt:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
```

7.3.2 Navigation Bar & Table View

Die *Navigation Bar* wird, wenn vorhanden, unter der *Status Bar* angezeigt. Sie ist zum Beispiel bei den Apps *Mail* oder *Einstellungen* zu sehen. Oft wird sie im Zusammenspiel mit einem *Table View* verwendet.

Ein Tipp auf eine Tabellenzelle führt dann einen animierten Drill-Down in eine tiefer gelegene Hierarchieebene aus. Die *Navigation Bar* enthält nach dem Drill-Down in der linken Ecke einen Pfeil mit dem Titel des vorherigen View. Ein Klick auf den Pfeil führt die Navigation in umgekehrter Richtung durch.

Die folgende Abbildung illustriert das Navigationsverhalten:

Bild 7.2: Drill-Down mit Navigation Bar und Tabelle

7.3.3 Tool Bar

Die *Tool Bar* sitzt am unteren Bildschirmrand und erlaubt das Ausführen von Aktionen im aktuellen Kontext, also zum angezeigten View. Zu diesem Zweck können der *Tool Bar* Items mit Icons hinzugefügt werden.

Bild 7.3: Tool Bar mit Items

7.3.4 Tab Bar

Alternativ zur *Tool Bar* kann am unteren Bildschirmrand eine *Tab Bar* (vergleichbar mit Karteikartenreitern) eingeblendet werden. Die darauf enthaltenen Tabs werden benutzt, um zwischen verschiedenen Sichten oder Modi der Anwendung umzuschalten. Nach einem Klick auf einen Tab bleibt dieser hervorgehoben. Im Content-Bereich darüber wird der dazugehörige View angezeigt.

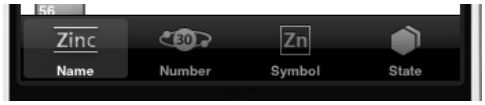


Bild 7.4: Die Tab Bar

7.3.5 Alert

Alerts sind kleine Meldungen, die modal und zentriert über einem View angezeigt werden. Sie sollten für Meldungen unerwarteter Zustände (keine Netzwerkverbindung oder ähnliches) verwendet werden. Alerts sind also nicht Teil des normalen Workflows und sollten nicht als Reaktion auf einen Button-Klick angezeigt werden. Für diesen Fall gibt es die als Nächstes zu besprechenden *Action Sheets*. Der *Alert* kann einen Text und einen oder zwei Buttons, zum Beispiel *OK* und *Abbrechen*, enthalten.

Bild 7.5: Alerts

7.3.6 Action Sheet

Action Sheets bieten für eine auszuführende *Tool Bar*-Aktion Ausführungsvarianten an. Für eine Aktion »Export« könnte das Action Sheet zum Beispiel Formatalternativen wie »Excel« oder »CSV« enthalten. Eine weitere Verwendungsmöglichkeit sind Sicherheitsabfragen im Falle eines *Tool Bar*-Buttons für das Löschen von Daten. Action Sheets werden immer vom unteren Bildschirmrand heraus nach oben animiert und legen sich über den View. Sie sind ebenfalls modal.

Bild 7.6: Action Sheets

7.3.7 Modale Views

Modale Views sind komplette Views die den ganzen darunter befindlichen View verdecken. Sie werden in der Standardeinstellung ebenfalls vom unteren Bildschirmrand hochgefahren. Wie das Action Sheet dient der modale View dazu, weitere Informationen vom Benutzer einzuholen. Während das Action Sheet nur die Auswahl eines der Buttons erlaubt, kann ein modaler View zum Beispiel Textfelder oder andere Controls enthalten.

7.3.8 Und der ganze Rest

Innerhalb des Content View wird oft eine Tabelle mithilfe eines *Table View* dargestellt. Neben (oder auch innerhalb) der Tabelle können aber auch eine Vielzahl weiterer Komponenten wie Buttons oder Textfelder verwendet werden, die hier nicht alle besprochen werden sollen.

Einen guten Überblick bietet die App »UICatalog«, die zur Sample-Sammlung von Apple gehört. Sie kann nach dem Download in Xcode geöffnet und anschließend im Simulator gestartet werden.

Bild 7.7: Die App »UICatalog«

7.4 Application Definition Statement & Features

Gerüstet mit dem frisch erworbenen Wissen über das Look & Feel von iPhone Apps beginnen wir mit dem Konzept für eine eigene Anwendung. Dabei muss zunächst die Frage geklärt werden, welchem Zweck die Anwendung dient.

Wichtig ist an dieser Stelle zu erkennen (und auch zu akzeptieren), dass mobile Anwendungen nicht die Vielzahl von Leistungsmerkmalen haben können wie etwa Desktop-Anwendungen. Das kleinere Display, das typische Nutzerverhalten (»beim Warten auf die U-Bahn mal schnell das Wetter checken«), die beschränkte Akkulaufzeit und viele Charakteristika mehr erzwingen eine Fokussierung auf ein Kernthema.

Das heißt nicht, dass mobile Anwendungen weniger leistungsfähig als ihre Desktop-Pendants sind. Das Einsatzgebiet ist einfach ein anderes. Niemand nimmt seinen Desktop-Rechner mit ins Auto, um sich an ein Fahrziel navigieren zu lassen. Im Umkehrschluss ergibt es keinen großen Sinn, Photoshop oder Word in all ihrer Leistungsvielfalt auf ein kleines mobiles Gerät zu bringen.

Das Kernthema für die App sollte schriftlich fixiert werden und als Richtlinie für die Konzeption und spätere Entwicklung dienen. Man spricht hier auch von einem *Application Definition Statement*. Das Application Definition Statement für die Anwendung, deren Entwicklung wir mit diesem Buch begleiten, lautet:

> »**Einfach zu bedienende mobile Zeiterfassung für Freiberufler**«

Alle Features, die in der Anwendung realisiert werden sollen, müssen sich am Application Definition Statement messen. Ist es sinnvoll, im Hinblick auf eine einfach zu bedienende mobile Zeiterfassung für Freiberufler dieses oder jenes Feature zu implementieren? Das Statement wirkt wie ein Filter und soll dabei helfen, sich auf das Wesentliche zu konzentrieren.

Anders als bei Desktop-Anwendungen, die gerne mit der Zahl ihrer Features werben, ist auf mobilen Geräten schlicht und einfach nur Platz für das Wesentliche. Vergleichen Sie Ihre Anwendung mit einem Werkzeugkasten. Natürlich wäre es toll, im Urlaub immer den gesamten Werkzeugkasten dabei zu haben. Aber aus Platzgründen können Sie nur das Schweizer Offiziersmesser mitnehmen. Sie erwarten nicht, dass sich daran beispielsweise eine Lötlampe befindet. Aber die wichtigsten Tätigkeiten sollten sich auch mit der mobilen Version des Werkzeugkastens erledigen lassen.

Seien Sie also eher sparsam mit den Leistungsmerkmalen, weniger ist mehr. Ziel ist eine in sich geschlossene, funktionierende Lösung und nicht eine Sammlung von Features zu einem Thema. Konzentrieren Sie sich auf die Features, die für die Mehrzahl der Benutzer am wichtigsten sind und die sich in geeigneter Weise auf ein mobiles Gerät bringen lassen.

Bei einer Portierung einer Web- oder Desktop-Anwendung sollte man besonders darauf achten, ob eventuell Fähigkeiten des iPhones genutzt werden können, die auf einem Desktop-Rechner gar nicht vorhanden sind. Beispielsweise sind dies:

- Kenntnis der aktuellen Position
- Kamera
- Beschleunigungssensor
- Kompass
- Mikrofon

Bei der Suche nach einer neuen Wohnung geben Sie bei einer Webanwendung Ihre Adresse ein, um Angebote in der Nähe zu finden. Beim iPhone sollte diese Funktion natürlich durch die automatische Bestimmung der Position erfolgen. Bei der Auswahl der Features sollten also immer auch die technischen Möglichkeiten von iPhone, iPod touch oder iPad bedacht werden.

Welche Features hat nun eine einfach zu bedienende mobile Zeiterfassung für Freiberufler? Ein erstes Brainstorming und ein Blick auf die Features von kommerziellen Weblösungen bringen einige Ideen hervor:

Bild 7.8: Ideen für die mobile Zeiterfassung

Jetzt geht's ans Filtern.

Wir verzichten auf jeden Fall auf eine Profilverwaltung und einen Kundenzugang. Beides sind typische Webanwendungs-Features, die auf einem mobilen Gerät keinen großen Sinn ergeben. Es wird also in unserem Fall immer der Besitzer des iPhones der Benutzer sein. Die Ideen »Spesenmanagement« und »Urlaubsplanung« sind zu weit weg von der Kernfunktionalität. Eine einfach zu bedienende mobile Zeiterfassung kann darauf sicherlich verzichten. Grafische Reports wären toll, sind aber bestimmt schwierig zu realisieren und auf dem kleinen Display eventuell nicht sonderlich hilfreich. Wir behalten sie vorerst im Hinterkopf. Der Rest der Features scheint unverzichtbar:

Bild 7.9: Ausgewählte Features

7.5 Objektmodell

Nachdem die Anforderungen mindestens in Form einer Featureliste (womöglich aber auch als Use Cases) definiert sind, wird als Nächstes ein Objektmodell benötigt.

Die Featureliste gibt einen Hinweis darauf, welche Objekte existieren und welche Zusammenhänge zwischen ihnen bestehen.

Wichtigste Aufgabe unserer Anwendung ist das Erfassen von Zeiten. Wir benötigen also bestimmt ein Objekt `Zeit` mit Attributen wie `Datum` und `Dauer`.

Einer erfassten Zeit kann eine Leistung zugeordnet werden. Im Falle eines freiberuflichen Softwareentwicklers könnten das »Training«, »Entwicklung« oder »Coaching« sein. Die Leistung hat neben diesem Namen vor allen Dingen einen Stundensatz.

Außer der erbrachten Leistung kann einer erfassten Zeit ein Projekt zugeordnet werden. Dieses hat ebenfalls einen Namen und eventuell ein Budget (in Form von Personentagen oder eines Betrags).

Das Projekt schließlich wird bei einem oder für einen Kunden ausgeführt.

Im *Core Data*-Datenmodell wird man später erkennen, dass alle Beziehungen optional sind. Für das User Interface Design ist das noch nicht relevant.

Die Attribute und Beziehungen der Objekte sind in der folgenden Abbildung dargestellt:

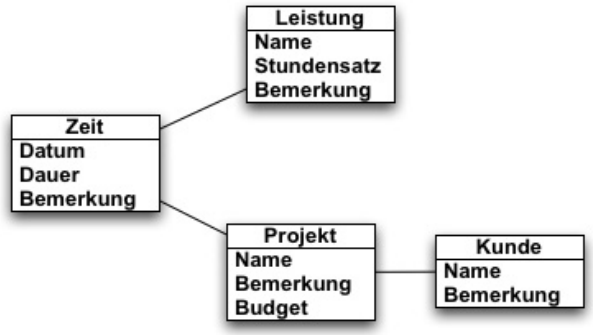


Bild 7.10: Das Objektmodell

Das Objektmodell hat Auswirkungen auf die Designs der einzelnen Screens und auf das Navigationsmodell. Außerdem ist es Grundlage für das spätere *Core Data*-Datenmodell, und damit auch für die zu erzeugenden Klassen.

7.6 Navigationsmodell & Skizzen

Das Navigationsmodell befasst sich mit der Frage, wie die einzelnen Screens der Anwendung miteinander verbunden sind, also wie der Anwender vom einen zum nächsten kommt.

Für hierarchische Zusammenhänge ist, wie erwähnt, die Navigation über die Kombination *Table View Controller* und *Navigation Controller* eine gute Wahl.

Problematisch wird es mit diesem Ansatz, wenn sehr viele Hierarchieebenen existieren. Ein Home-Button zum Sprung auf das Hauptmenü würde das »mentale Navigationsmodell« des Anwenders komplett zerstören und kommt deshalb nicht infrage. Folglich müssen immer alle Ebenen durchlaufen werden: Von der untersten Stufe einer Kategorie zurück bis zur höchsten Ebene und dann absteigend in eine andere Kategorie können da schon einige Klicks zusammenkommen.

Abhilfe kann die Verwendung der *Tab Bar* für die Top-Level-Hierarchie schaffen. Anstelle eines Hauptmenüs mit Einträgen wie in unserem Fall *Kunden*, *Projekte* usw. lassen sich diese Bereiche per Tab erreichen. Innerhalb der Tabs finden sich dann Screens mit der Navigation über *Table View Controller* und *Navigation Controller*. Neben dem Einsparen einer Ebene hat die Lösung den Charme, dass die Navigation in den einzelnen Bereichen beim Wechsel erhalten bleibt. Der Tab wird beim nächsten Betreten in dem Zustand vorgefunden, in dem er verlassen wurde.

Ein Beispiel für ein solches Navigationsmodell ist die Anwendung *App Store*. Wenn man im Tab *Highlights* eine App auswählt und durch Klick auf die Zelle zu den Infos wech-

selt, bleibt dieser Details-Screen erhalten, auch wenn man sich zwischendurch die Top 25 angesehen hat.

Bild 7.11: App mit Tab Bar

Was auf den ersten Blick nach einer brauchbaren Lösung aussieht, hat aber einen kleinen Schönheitsfehler. Der untere Bildschirmrand ist nun durch die Tab Bar geblockt und kann keine Tool Bar mehr aufnehmen. Technisch könnte man womöglich beide Bars anzeigen, optisch wäre es in jedem Fall eine Zumutung.

Verwendet man also die Tab Bar, müssen Aktionen für den betreffenden Screen entweder im Content-View selber oder aber in der *Navigation Bar* am oberen Bildschirmrand untergebracht werden. Die *Navigation Bar* wiederum ist durch den Titel, das Back-*Navigation Bar Item* und den oft benötigten *Edit/Done*-Button schon ziemlich voll.

Eine Lösung für alle denkbaren Fälle gibt es nicht. Welche Variante (Tab Bar vs. Table View/Navigation Controller) für die Top-Level-Hierarchie nun die richtige ist, hängt von den weiteren Anforderungen der Anwendung ab. Möglich ist beispielsweise auch ein Ausblenden der Tab Bar beim Drill-Down in ein Detail.

7.6.1 Hauptmenü

Für die Zeiterfassung soll nicht auf die *Tool Bar* verzichtet werden, das Hauptmenü wird als Tabelle und nicht in Form von Tabs realisiert.

Wie die Featureliste und das Objektmodell erkennen lassen, kann die geplante Anwendung in die vier großen Bereiche *Kunden*, *Projekte*, *Leistungen* und *Zeiten* unterteilt wer-

den. Diese Bereiche sind die Einträge im Hauptmenü. Für jeden Bereich wird jeweils CRUD-Funktionalität benötigt. CRUD steht für *Create*, *Read*, *Update* und *Delete*, also die gängigen Aktionen in datenzentrierten Anwendungen.

Um die ersten Visionen des User Interface zu Papier zu bringen, empfehlen sich auf jeden Fall Bleistiftskizzen. Fehler sind schnell wegradiert, und insgesamt ist man deutlich schneller als mit Unterstützung einer Software. Legen Sie einfach Ihr iPhone auf ein Blatt Papier und ziehen mit einem Bleistift eine Linie rundherum.

Entscheidend ist nicht die künstlerische Umsetzung des Entwurfs, sondern vielmehr, daran zu denken, dass alle geplanten Aktionen und Features ihren Platz finden. Dieser Punkt ist wirklich wichtig. Das nachträgliche Zufügen von Funktionen kann bei der Mobile-Entwicklung wegen des knappen Platzangebots zu einem echten Problem werden.

Im Top-Level-Menü gestaltet sich die Aufgabe einfach. Ein Klick auf einen Menüpunkt führt zur Liste von Kunden, Projekten usw. Ein Info-Dialog soll den Benutzer über Version und Entwickler informieren. Die oberste Hierarchieebene des *Navigation Controller* (auf der wir uns gerade befinden) bietet auch einen schönen Platz zur Anzeige des Applikationsnamens in der Mitte der *Navigation Bar*. Bei Bedarf kann hier auch ein Logo oder eine andere Grafik angezeigt werden.

Die folgende Abbildung zeigt das Hauptmenü unserer Anwendung *Chronos* als Skizze:

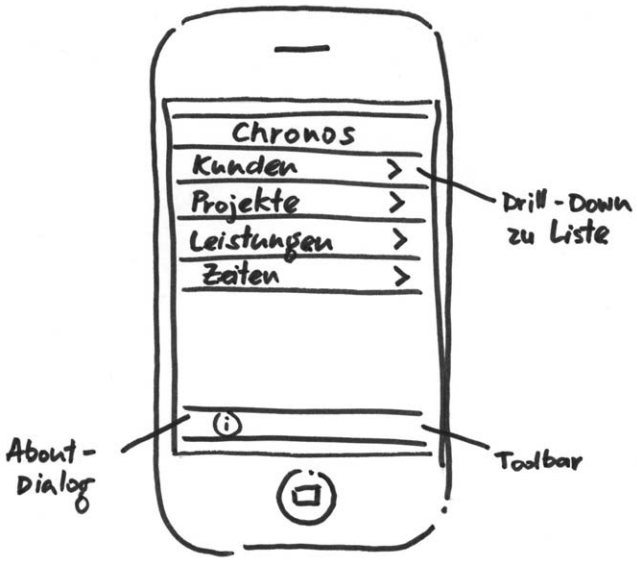


Bild 7.12:
Skizze des Hauptmenüs

Der Entwurf für das Hauptmenü ist damit vorerst abgeschlossen. Sowohl das Menü als auch die Tool Bar bieten reichlich Platz für Erweiterungen oder womöglich übersehene Funktionen.

Betrachten wir die nächste Ebene am Beispiel der Leistungen.

7.6.2 Leistungsliste

Von der Liste der Leistungen aus, die man durch Auswahl des Menüpunks in der obersten Ebene erreicht, sollen folgende Aktionen ausführbar sein:

- Navigation zurück zum Hauptmenü
- Bearbeiten einer Leistung
- Löschen einer Leistung
- Hinzufügen einer Leistung

Die Navigation zurück zum Hauptmenü ist Aufgabe des Navigation Controller, und der dafür zuständige Button hat links oben in der Navigation Bar seinen festen Platz.

Gleich daneben findet sich der Name des ausgewählten Menüpunkts. Er ist wichtig, um dem Benutzer anzuzeigen, an welcher Stelle der Hierarchie er sich gerade befindet.

Auf der rechten Seite der Navigation Bar ist der Button für den Edit-Mode eingeplant. Der Modus übernimmt die Visualisierung des Löschens von Datensätzen, in unserem Fall von Leistungen.

Das Bearbeiten einer Leistung erfolgt durch Drill-Down in den Detail-Screen. Er wird durch Klick auf eine Tabellenzelle ausgeführt.

Schließlich muss noch das Hinzufügen von Leistungen untergebracht werden. Innerhalb der Tabellenzellen ergibt das keinen Sinn, da sich die Aktion ja nicht auf eine existierende Leistung bezieht. Eine Lösung wäre ein Button im Content-Bereich unterhalb der Tabelle, also im Tabellen-Footer. Der Footer verschiebt sich allerdings beim Scrollen und kann dadurch bei vielen Datensätzen in der Tabelle auch außerhalb des sichtbaren Bereichs liegen. Glücklicherweise haben wir uns ja für die Tool Bar entschieden und brauchen nicht weiter zu grübeln.

In der Tool Bar ist allerhand Platz für Aktionen, die sich auf den aktuellen Screen beziehen. Auf der rechten Seite der Leiste soll ein Button mit einem »+«-Zeichen das Zufügen von Leistungen über einen modalen *View Controller* erlauben. Modale View Controller können ergänzend zur bestehenden Navigation, ähnlich wie ein Dialog, eingeblendet werden. Sie legen sich über den gerade sichtbaren View.

Die folgende Abbildung zeigt die Skizze für die Leistungsliste:

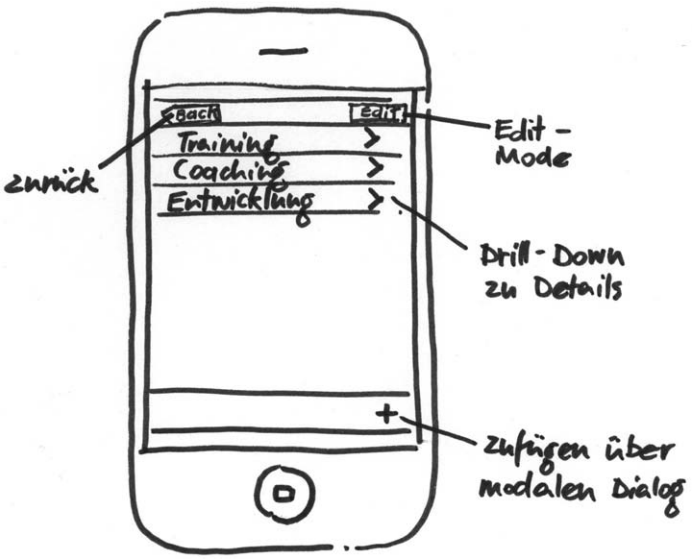


Bild 7.13: Skizze der Leistungsliste

7.6.3 Leistungsdetails

Betrachten wir nun den Leistungsdetail-Screen.

Hier müssen folgende Aktionen ausführbar sein:

- Navigation zurück zur Liste
- Bearbeiten der Attribute wie Name, Stundensatz, Bemerkung

Die Navigation zurück zur Leistungsliste ist wiederum Aufgabe des *Navigation Controller* und damit »gesetzt« in der oberen linken Ecke.

Die Bearbeitung der einzelnen Attribute könnte entweder direkt auf dem Detail-Screen oder aber auf einem separaten Screen pro Attribut erfolgen. Für beide Varianten gibt es schöne Beispiele im App Store.

Die App »Post mobil« setzt auf die Bearbeitung direkt im Detail-Screen. Wird in das Textfeld *Straße* »geklickt« (man spricht beim iPhone eigentlich eher von einem *Tap*), fährt die einblendbare Tastatur hoch.

Bild 7.14: Die App »Post mobil«

Das Problem dabei ist, dass die Tastatur die Hälfte des Displays verdeckt und damit eventuell Daten nicht erreichbar sind. Der Entwickler muss also dafür sorgen, dass sich trotz der Tastatur noch alle Attribute bearbeiten lassen.

Bei »Post mobil« wird das Problem mit den Buttons *Zurück* und *Weiter* gelöst. Sie erlauben die Navigation von Attribut zu Attribut durch Verschieben des Views unterhalb der Tastatur.

Vorteil dieser Variante ist, dass ein einziger Tap genügt, um von der Bearbeitung eines Attributs in die Bearbeitung des nächsten Attributs zu gelangen.

Die Alternative dazu lässt sich in der App »Kontakte« testen.

Bild 7.15: Die App »Kontakte«

Auch die Attribute sind bei dieser Anwendung im Detail-Screen als Tabelle realisiert. Bei jedem Tap auf ein Attribut wird ein separater View zur Bearbeitung eingeblendet.

Diese Variante ist insgesamt ein wenig flexibler. Zum Beispiel bietet sie Platz für erläuternde Texte oder weitere Controls im separaten View. Nachteilig ist, dass hier zwei Taps benötigt werden, um von Attribut zu Attribut zu kommen. Der separate View muss zunächst verlassen und anschließend kann das nächste Attribut zur Bearbeitung ausgewählt werden. Erbsenzählerei? Denken Sie an die wenigen Sekunden, die Ihrer App zur Verfügung stehen, um ihre Aufgaben zu erfüllen ...

Wir entscheiden uns dennoch für Variante zwei. Der Screen zur Bearbeitung eines Attributs soll also vom Leistungsdetail-Screen nach Aktivieren des Edit-Modes und anschließender Auswahl des entsprechenden Attributs angezeigt werden.

Der Button für den Edit-Mode wird, wie in der zuvor besprochenen Leistungsliste, auf der rechten Seite des Navigation Controller angezeigt.

Die nächste Abbildung zeigt die aus den Überlegungen resultierende Skizze für den Leistungsdetail-View:

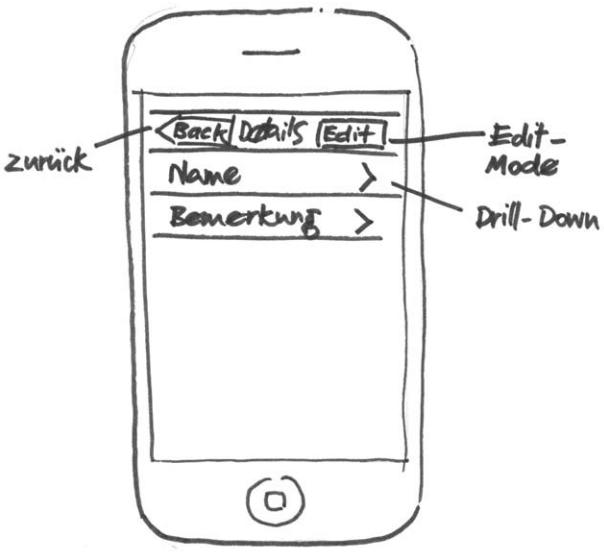


Bild 7.16: Die Leistungsdetails der Applikation

Für den Screen zum Bearbeiten der Attribute verzichten wir auf eine Skizze. Er wird sich an dem rechten der beiden Screenshots für die Kontakte-App orientieren. Im oberen Bereich befindet sich ein Textfeld zur Eingabe und Anzeige des Wertes. Darunter wird die Tastatur eingeblendet. Bei Attributen, die aus einer Liste ausgewählt werden können, wird stattdessen ein Picker angezeigt. Der Screen kann über *Abbrechen* oder *Sichern* verlassen werden.

7.6.4 Zeitenliste

Die Konzeption für die anderen Bereiche (Projekte, Leistungen, Zeiten) erfolgt weitgehend analog bzw. kann entfallen, weil offensichtlich ist, dass die Implementierung analog erfolgen kann. Lediglich der Listen-View für die Zeiten ist ein wenig aufwendiger und soll noch kurz besprochen werden.

Neben den typischen CRUD-Funktionalitäten kommen hier noch folgende Anforderungen hinzu:

- Sortieren der Liste nach verschiedenen Kriterien
- Filtern der Liste (ebenfalls nach verschiedenen Kriterien)
- Erzeugen von Reports oder Exports

Das Sortieren der Liste soll über ein *Segmented Control* direkt auf dem Screen erfolgen. Denkbare Positionen für das Control sind in der Mitte der Navigation Bar (statt des Titels) oder aber in der Tool Bar. Der Verlust des Titels würde schmerzen, soll er doch

eigentlich bei der Orientierung helfen. In diesem Fall scheint es trotzdem der »richtige« Weg zu sein, weil die Sortierfunktion möglichst über den zu sortierenden Daten stehen sollte.

Das Filtern der Daten und das Erzeugen von Reports soll mit *Action Sheets* oder modalen *View Controllern*, also auf separaten Views, abgehandelt werden. Es wird also nur Platz für die Anzeige der zugehörigen Buttons benötigt. Davon haben wir auf der Tool Bar noch reichlich.

Die komplette Skizze für die Zeitenliste sieht folgendermaßen aus:

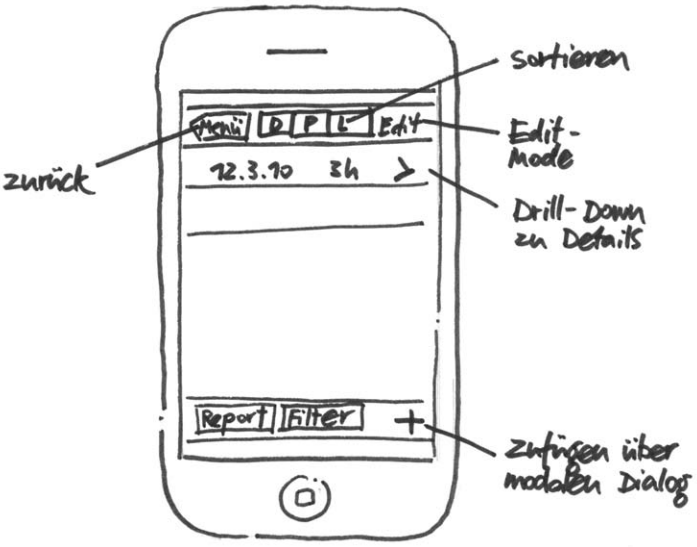


Bild 7.17: Skizze für die Zeitenliste

Das Navigationsmodell und die Skizzen der Screens sind damit fertiggestellt. Es muss zu diesem Zeitpunkt klar sein, wie man zu den einzelnen Views gelangt und was passiert, wenn auf irgendeinen Button geklickt wird. Die Anwendung kann also in Gedanken komplett bedient werden.

7.7 Bessere Mock-ups

Für eigene Anwendungen werden die Bleistiftskizzen vollkommen genügen. Letztlich möchten wir ja eine funktionierende App und nicht tolle Bilder in den Händen halten. Soll allerdings Software für dritte Personen (Freunde oder gar Kunden) entwickelt werden, muss womöglich die Qualität der Vorschaubilder erhöht werden.

Wenn dies der Fall ist, kann das hervorragende Programm *OmniGraffle* [URL-GRAFFLE] zum Anfertigen von Mock-up-Bildern empfohlen werden. *OmniGraffle* entspricht in der Funktionalität etwa dem bekannten Microsoft-Produkt *Visio*. Für das

Zeichen-Tool stehen mehrere iPhone-Erweiterungen (sogenannte »Stencils«) zur Verfügung.

Folgendes Mock-up wurde mit *OmniGraffle* und der iPhone-WebApp-WireFrame-Erweiterung angefertigt:

Bild 7.18: OmniGraffle Mock-up

Eine Alternative zu *OmniGraffle* oder anderen Zeichenprogrammen kann der Interface Builder sein.

Wie wir gesehen haben, ist das Anfertigen der Oberflächen auch nicht schwieriger als das Bedienen eines Zeichenprogramms. Mit ein wenig Übung im Interface Builder ist das durchaus eine realistische Option. Die fertige Mock-up-Oberfläche kann im Interface Builder über den Menüpunkt *File > Simulate Interface* im Simulator angezeigt und beispielsweise mittels der Anwendung *Bildschirmfoto* in einen Screenshot umgewandelt werden.

Diese Vorgehensweise hat den schönen Nebeneffekt, dass man die fertigen Oberflächen auch für die spätere Realisierung verwenden kann.

8 Projektstart

Das Konzept für die Anwendung steht. In diesem Kapitel wird nun endlich mit der Entwicklung begonnen. Dabei stehen zunächst einige administrative Tätigkeiten wie das Einrichten der Versionsverwaltung und das Erzeugen eines Entwicklungsprofils zum Start der App auf der Hardware im Vordergrund.

8.1 Versionsverwaltung

Grundsätzlich ist es bei nahezu allen Softwareprojekten sinnvoll, eine Versionsverwaltung zu verwenden. Selbst wenn man allein am Projekt arbeitet, ist es ein unschätzbarer Vorteil, nach ein paar unglücklichen Änderungen auf den letzten funktionierenden Stand zurückfallen zu können.

Für das Buchprojekt wird die freie Software *Subversion* verwendet. Das Produkt ähnelt dem bekannten *CVS*, verfügt aber über ein paar weitere Features. Genannt sei die wichtige Möglichkeit, Dateien und Verzeichnisse zu verschieben oder umzubenennen, ohne dabei die Historie zu verlieren.

Ab Mac OS 10.5 (*Leopard*) ist Subversion bereits vorinstalliert. Zur Überprüfung wechseln Sie in ein Terminalfenster und geben das Kommando `svn help` ein. Subversion meldet sich mit der Versionsnummer und einer Übersicht über die möglichen Kommandos.

Projekte werden in Subversion in *Repositories* gespeichert. Es handelt sich dabei um eine Art Datenbank für die Dateien. Zum Anlegen eines neuen Repository geben Sie folgenden Befehl ins Terminalfenster ein (der Pfad muss natürlich angepasst werden):

```
svnadmin create /Users/koller/Documents/svnrepository
```

Als Nächstes machen wir Xcode mit dem neuen Repository bekannt, um die integrierten Features für das Quellcode-Management nutzen zu können.

Unter *SCM > Repositories* findet sich in Xcode ein Repository-Browser. Durch einen Klick auf *Configure* in der Toolbar des Dialogs öffnet sich der SCM-Reiter aus den Xcode-Preferences. Dieser erlaubt via »+« das Zufügen von neuen Repositories. Neben dem Namen des Repository ist für eine lokal installierte Subversion-Instanz noch die Angabe des Pfades erforderlich:

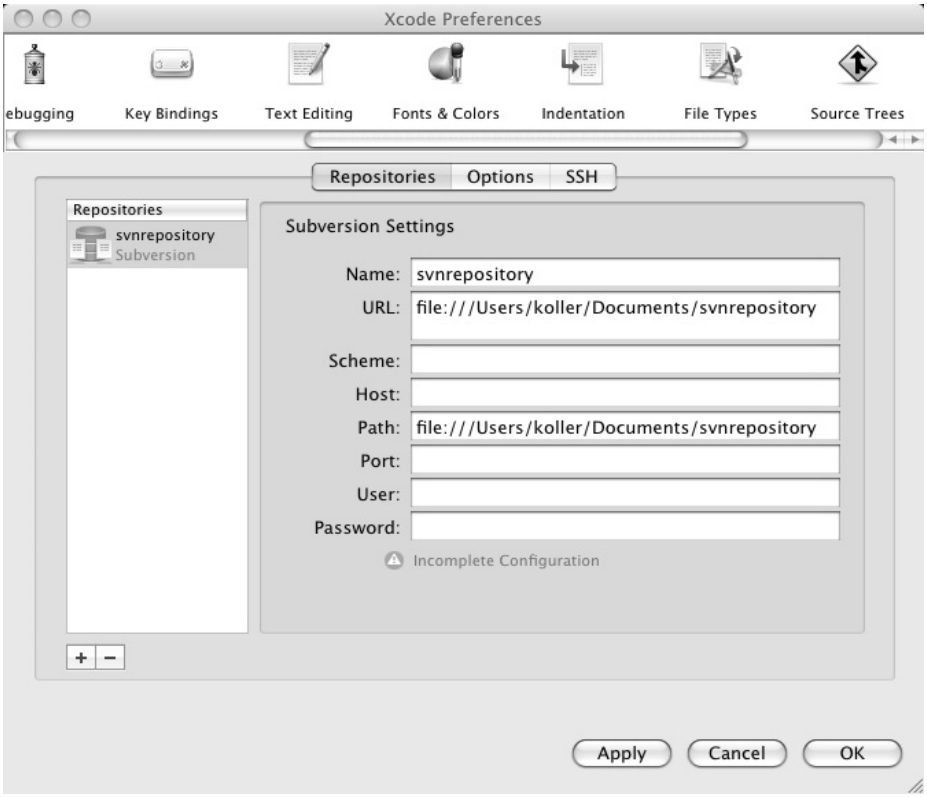


Bild 8.1: Subversion in Xcode einrichten

Nach dem Anlegen des Repository kann der Inhalt mit dem Browser untersucht werden.

8.2 Generieren des Projekts

Wir beginnen das Erstellen unseres Projekts mit dem Anlegen eines Ordners *chronos*, der den Projektcode aufnehmen soll. *Chronos* ist das griechische Wort für Zeit und soll als Arbeitstitel für das Projekt dienen. Da als Versionsverwaltung Subversion verwendet wird, benötigen wir die in diesem Produkt üblichen Unterordner `branches` (alternative Entwicklungspfade), `tags` (benannte Versionsstände) und `trunk` (Hauptentwicklungslinie).

Sind die Ordner im Dateisystem angelegt, wird Xcode gestartet und mit *File > New Project...* der Assistent zum Anlegen neuer Projekte ausgeführt.

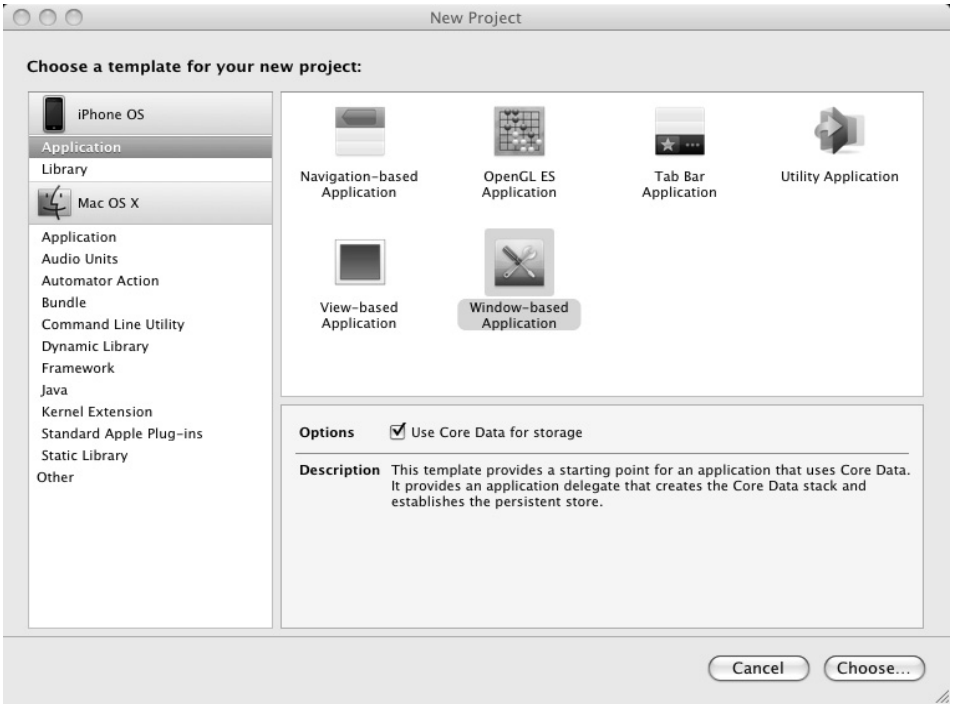


Bild 8.2: Projekte mittels einer Vorlage erzeugen

Als Vorlage für das Projekt wird *Window-based Application* gewählt und außerdem die Checkbox für die Benutzung von *Core Data* als Persistenzschicht gesetzt. Mit *Choose...* gelangt man zur Auswahl des Projektverzeichnisses (in unserem Fall ein beliebiger Ordner, zum Beispiel der Schreibtisch) und des Projektnamens (*chronos*). Anschließend wird das Xcode-Fenster erst einmal beendet und der Inhalt von `Schreibtisch/chronos` in den richtigen Projektordner `chronos/trunk` verschoben. Der nun leere Ordner auf dem Schreibtisch kann gelöscht werden.

Wenn alles geklappt hat, sollte folgende Verzeichnisstruktur auf der Festplatte existieren:

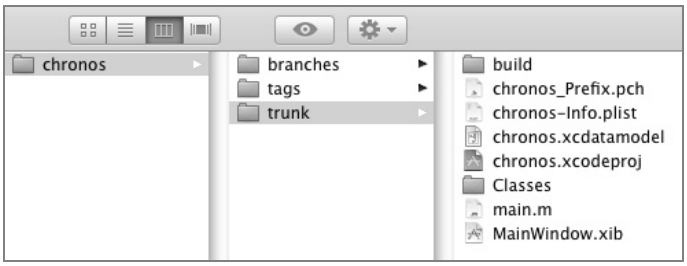


Bild 8.3: Die Verzeichnisstruktur

Der Projektordner *chronos* wird im folgenden Schritt dem Subversion-Repository zugefügt. Das kann zum Beispiel von einem Terminalfenster aus erfolgen:

```
svn import /Users/koller/Documents/Projekte/chronos
   file:///Users/koller/Documents/svnrepository/chronos -m "initial"
```

Ist das Projekt erfolgreich importiert, darf der Ordner `chronos` mitsamt Inhalt gelöscht werden. Ab jetzt wird auf einer *Working Copy* gearbeitet.

Man erhält die Kopie durch:

```
svn checkout file:///Users/koller/Documents/svnrepository/chronos
   /Users/koller/Documents/Projekte/chronos
```

Durch Doppelklick auf `chronos.xcodeproj` wird das Projekt in Xcode geöffnet. Über den Menüpunkt *SCM > Configure SCM for this Project* wird Xcode mitgeteilt, dass das Projekt unter Sourcecode-Verwaltung steht. Hier ist in erster Linie die Auswahl des SCM-Repository wichtig.

8.3 Projektstruktur

Ein Vergleich der Dateien und Verzeichnisse im Dateisystem-Projektordner mit den Gruppen und Dateien in Xcode zeigt, dass einige Unterschiede bestehen. Xcode visualisiert nicht, wie andere Entwicklungsumgebungen, das Projektverzeichnis auf dem Datenträger, sondern arbeitet mit einer virtuellen Verzeichnisstruktur.

Im Folgenden werden die vom Assistenten erzeugten Dateien der Projektvorlage besprochen.

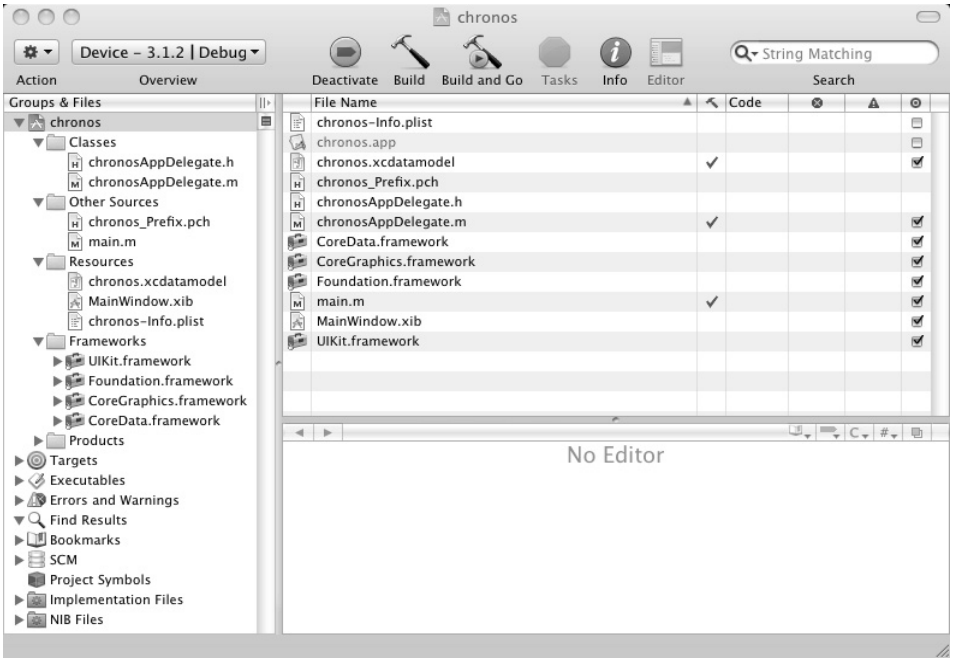


Bild 8.4: Das generierte Projekt

8.3.1 chronos-Info.plist

Dateien mit der Endung `.plist` sind spezielle Property-Dateien in einem von Apple festgelegten XML-Format. Im Xcode-Editor-Fenster ist die XML-Struktur nicht sichtbar. Man kann die Datei aber in einem Texteditor öffnen, um das Format anzusehen. Der Vorteil von `.plist`-Dateien liegt in der Verfügbarkeit spezieller Methoden, um die Dateien in entsprechende Objekte (und umgekehrt) umzuwandeln. Wie man sieht, enthält die Datei in Key-Value-Paaren für das Projekt wichtige Informationen, wie zum Beispiel den Namen der Haupt-Nib-Datei:

Key	Value
Information Property List	(12 items)
Localization native development region	English
Bundle display name	${PRODUCT_NAME}
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.yourcompany.${PRODUCT_NAME:rfc10
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle version	1.0
LSRequiresIPhoneOS	☑
Main nib file base name	MainWindow

Bild 8.5: Key-Value-Paare in der Datei `chronos-Info.plist`

Im weiteren Verlauf werden immer mal wieder Änderungen an der Datei `chronos-Info.plist` vorgenommen.

8.3.2 chronos.xcdatamodel

Die Datei `chronos.xcdatamodel` enthält das *Core Data*-Datenmodell. Ein Doppelklick auf die Datei öffnet den Editor, der das Anlegen von Diagrammen erlaubt, die Entity-Relationship-Modellen ähneln. Im Kapitel über Core Data wird von dem Editor Gebrauch gemacht.

8.3.3 chronos_Prefix.pch

Um nicht in jeder Sourcecode-Datei die benötigten Frameworks *Foundation*, *UIKit* und *Core Data* importieren zu müssen, sind diese in der Datei `chronos_Prefix.pch` hinterlegt und werden automatisch in jede Datei importiert.

8.3.4 main.m

Die Datei `main.m` braucht im Normalfall nicht abgeändert zu werden. In ihr ist die Implementierung der `main()`-Funktion zu finden. Sie wird vom System aufgerufen, um die App zu starten:

```
int main(int argc, char *argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   int retVal = UIApplicationMain(argc, argv, nil, nil);
   [pool release];
   return retVal;
}
```

Interessant ist der `AutoreleasePool`, der vor dem Aufruf der Haupt-Eventschleife erzeugt und vor dem Beenden der App wieder freigegeben wird. Wie schon besprochen, landen in diesem Pool alle Objekte, die eine `autorelease`-Nachricht erhalten haben.

8.3.5 chronosAppDelegate.h

Die Klasse, die auf `AppDelegate` endet, ist der Einstiegspunkt für den Entwickler. In ihr werden zahlreiche Lebenszyklusmethoden implementiert, die vom Application-Objekt zu gegebener Zeit aufgerufen werden. Im Header-File ist zu erkennen, dass zur Realisierung dieser Funktionalität das Protokoll *UIApplicationDelegate* implementiert wird. Es enthält zahlreiche optionale Methoden, die bei Bedarf mit Leben gefüllt werden können. Vielleicht versuchen Sie zur Übung mal das Protokoll in der Dokumentation aufzuspüren.

8.3.6 chronosAppDelgate.m

Hierbei handelt es sich um die Implementierung des `ApplicationDelegate`. Nach dem Generieren des Projekts ist erst eine der Lebenszyklusmethoden implementiert.

8.3.7 MainWindow.xib

`MainWindow.xib` ist die in der `.plist`-Datei angegebene Haupt-Nib-Datei, also eine Datei mit GUI-Informationen des Programms Interface Builder.

8.4 Erster Start

Die generierte Anwendung kann bereits im iPhone Simulator gestartet werden.

Dazu wird in Xcode unter *Project > Set Active SDK* (oder einfacher über die entsprechende Auswahlliste in der Toolbar) der Simulator ausgewählt und die App mit *Build and Go* gestartet. Das Ergebnis ist eher ernüchternd, im iPhone-Simulator ist lediglich eine weiße Fläche (der Hintergrund des Window) zu sehen.

Um die App auch auf dem echten mobilen Gerät starten zu können, ist noch einige Arbeit notwendig.

Ein erster naiver Versuch, das Programm einfach durch Setzen der *Overview*-Auswahlliste auf *iPhone Device* und anschließendes Betätigen der *Build and Go*-Schaltfläche auf dem Gerät zu starten, scheitert kläglich:

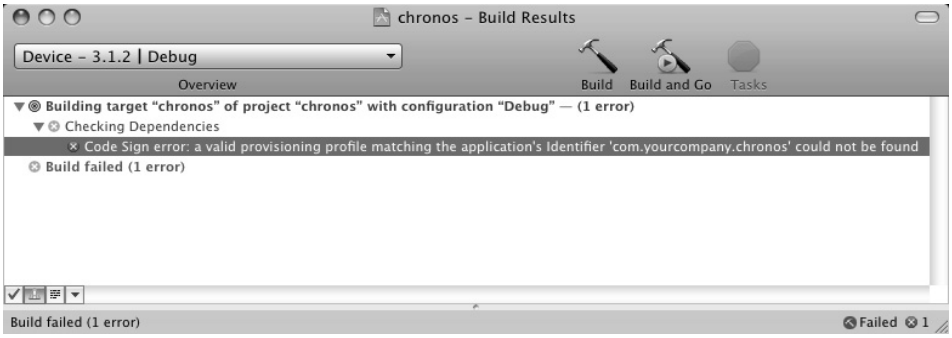


Bild 8.6: Fehlendes Profil

Wie uns die Fehlermeldung belehrt, wird zum Ausführen der Anwendung auf der echten Hardware ein *Provisioning Profile* benötigt. Dabei handelt es sich um eine Erlaubnis, eine bestimmte App auf einem bestimmten Gerät starten zu dürfen.

Man erstellt die Datei im *iPhone Developer Program Portal*, das über den gleichnamigen Link aus dem *iPhone Dev Center* erreicht werden kann. Wie schon in der Einleitung

erwähnt, ist dafür allerdings eine Mitgliedschaft im *iPhone Developer Program* Voraussetzung.

Bild 8.7: iPhone Developer Program Portal

Für das Erstellen der Erlaubnisdatei werden ein Zertifikat, eine *App ID* und ein registriertes *Device* benötigt, es gibt also einiges im Portal zu tun. Einige Bereiche im Developer Program Portal enthalten Karteikarten für *Development* und *Distribution*. Die folgenden Schritte in diesem Kapitel beziehen sich immer auf den Reiter *Development*.

8.4.1 Devices

Beginnen wir mit etwas Einfachem und fügen ein Gerät über den Link *Devices* und anschließend *Add Device* hinzu.

Neben einem Namen, über den das Gerät erkannt werden kann, wird die *Device ID* (*UDID*) benötigt. Der 40-stellige Hex-String kann über *iTunes* herausgefunden werden.

Dazu wird bei angeschlossenem Gerät in der Baumansicht auf der linken Seite des *iTunes*-Fensters das betreffende iPhone unter *Geräte* ausgewählt. Im Tab *Übersicht* klickt man dann auf die Seriennummer – voilà. Die Seriennummer verschwindet und macht der gesuchten *UDID* Platz. Sie kann nun über die Zwischenablage kopiert und in die Webseite eingetragen werden.

Bild 8.8: UDID in iTunes

Die *UDID* kann übrigens auch mit dem in Xcode enthaltenen Organizer ermittelt werden. Wenn allerdings die Geräte-ID eines Kunden oder Bekannten (zum Beispiel für einen Beta-Test) benötigt wird, ist die iTunes-Variante besser geeignet. Xcode ist schließlich im Gegensatz zu iTunes nicht auf jedem Mac-Rechner installiert und schon gar nicht auf Windows-Rechnern.

8.4.2 Zertifikat

Richtig aufwendig ist das Erstellen des Entwicklungszertifikats. Die Anwendung muss mithilfe eines Zertifikats signiert werden, um sicherzustellen, dass die App auch wirklich von der angegebenen Quelle (also von Ihnen) stammt.

Die ganze Prozedur beginnt mit dem Erstellen eines *Certificate Signing Request* in der Mac-Anwendung *Schlüsselbundverwaltung*. Über den Menüpunkt *Schlüsselpunktverwaltung > Zertifikatsassistent > Zertifikat einer Zertifizierungsinstanz anfordern...* wird der Assistent gestartet. Neben der Eingabe von Name und E-Mail-Adresse sollte auch der Radio-Button *Auf der Festplatte sichern* ausgewählt werden.

Bild 8.9: Ein Zertifikat anfordern

Im nächsten Schritt des Assistenten kann die Datei dadurch auf der Festplatte abgelegt werden.

Ist der Request erstellt, wird er über den Button *Request Certificate* im *Certificates* Bereich des Program Portals hochgeladen. Das angeforderte Zertifikat steht kurze Zeit später zum Download bereit.

Bild 8.10: Zertifikat bereit zum Download

Zertfikate werden in der Schlüsselbundverwaltung aufbewahrt und verwaltet. Ein Doppelklick auf das geladene Zertifikat öffnet das Programm und fügt das neue Zertifikat nach einer Abfrage hinzu.

Wie Sie dem Hinweis auf der Webseite entnehmen können, benötigen Sie neben dem gerade erstellten Zertifikat noch ein weiteres, das *WWDR*-Zertifikat von Apple. Glücklicherweise kann es über den Link auf der Seite *Certificates* einfach heruntergeladen und durch Doppelklick installiert werden.

Hat alles geklappt, sind in der Schlüsselbundverwaltung nun zwei neue Zertifikate enthalten: eines mit dem Namen *iPhone Developer: {Ihr Name}* und eines mit dem Namen *Apple Worldwide Developer Relations Certification Authority*. Ob das auch wirklich so ist, sollte vor dem Fortfahren unbedingt überprüft werden.

8.4.3 App ID

Als Nächstes wird auf der über den Link *App IDs > New App ID* erreichbaren Seite eine *App ID* vergeben. Es handelt sich dabei um einen eindeutigen Namen für die Anwendung. Die *App ID* besteht aus einem von Apple erzeugten Präfix namens *Bundle Seed Id* und einem *Bundle Identifier*, der vom Entwickler vergeben wird. Um Namenskonflikte zu vermeiden, empfiehlt Apple, den *Bundle Identifier* im umgekehrten Domain-Stil zu vergeben, also etwa `com.domainname.appname`. Da schon die *Bundle Seed Id* eindeutig ist, ist diese Empfehlung nicht ganz so wichtig. Außerdem kann auf der Seite noch eine Beschreibung zur Identifizierung der App im Portal eingegeben werden:

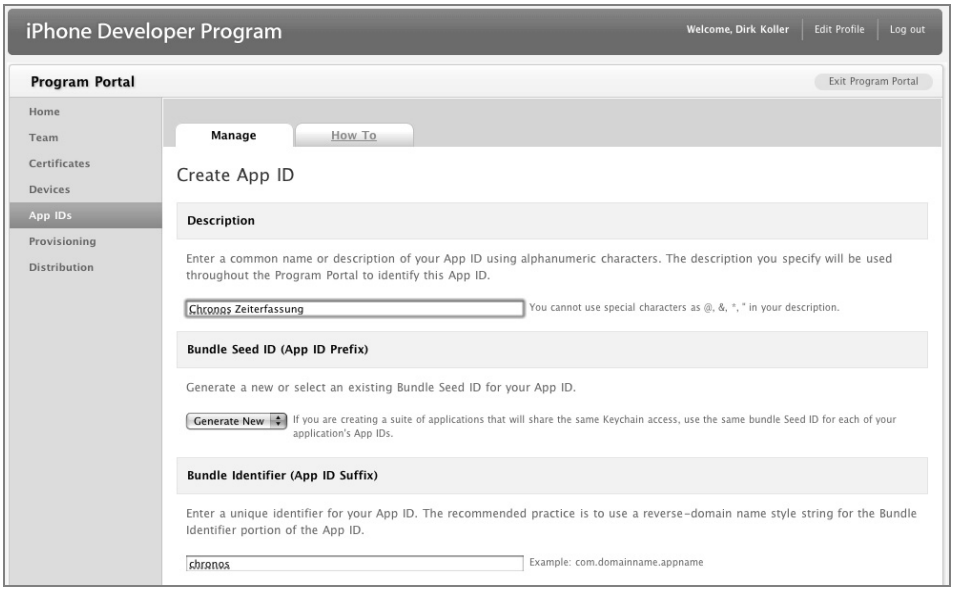


Bild 8.11: Die App ID erzeugen

8.4.4 Provisioning Profile

Sind Device, Zertifikat und App ID erstellt, darf endlich der Link *Provisioning* und anschließend auf dem Reiter *Development* der Button *New Profile* angeklickt werden. Das neue Profil bekommt einen Namen (zum Beispiel *Chronos Development Profile*) und ein Häkchen in der *Certificates*-Checkbox. Außerdem werden die zuvor angelegte App ID und das gewünschte Gerät ausgewählt:

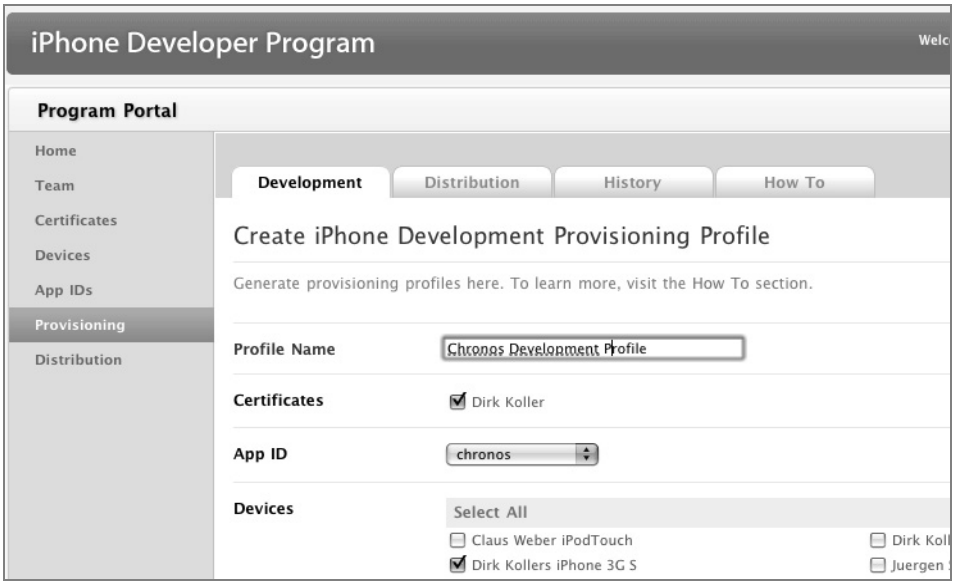


Bild 8.12: Ein Entwicklungsprofil anlegen

Das neue Profil kann nach der Fertigstellung heruntergeladen werden.

Danach muss es sowohl auf das iPhone gebracht als auch in Xcode registriert werden. Dazu öffnet man bei angeschlossenem iPhone in Xcode den Organizer (*Window* > *Organizer*) und zieht das Profil direkt aus dem Finder sowohl auf den iPhone-Eintrag (unter *Devices*) als auch auf *Provisioning Profiles* (unter *IPHONE DEVELOPMENT*).

Ob das Profil erfolgreich installiert wurde, kann im iPhone unter *Einstellungen* > *Allgemein* > *Profile* überprüft werden. Findet sich hier kein Eintrag, kann nicht fortgefahren werden und es muss zunächst der Fehler gesucht werden.

Bild 8.13: Entwicklungsprofil auf dem iPhone

Außerdem sollte nach dem Installieren nochmals der Organizer gecheckt werden. Das Profil muss sowohl unter *IPHONE DEVELOPMENT/Provisioning Profiles* als auch unter *DEVICES/{Ihr Gerät}* im Abschnitt *Provisioning* des Reiters *Summary* auftauchen. Die kleine Lampe neben dem Namen Ihres iPhones unter *Devices* muss grün leuchten. Ist das Lämpchen gelb oder rot, wird im Organizer eine Meldung angezeigt, die das Problem beschreibt.

Ist das Profil erfolgreich installiert, muss als Nächstes in der Debug-Konfiguration im Reiter *Build* der *Project Info* die *Code Signing Identity* aus der Auswahlliste ausgewählt werden:

Bild 8.14: Die Code Signing Identity auswählen

In der `Info.plist`-Datei ist schließlich der beim Erzeugen der *App ID* vergebene *Bundle Identifier* zu setzen, in unserem Fall also auf *chronos*:

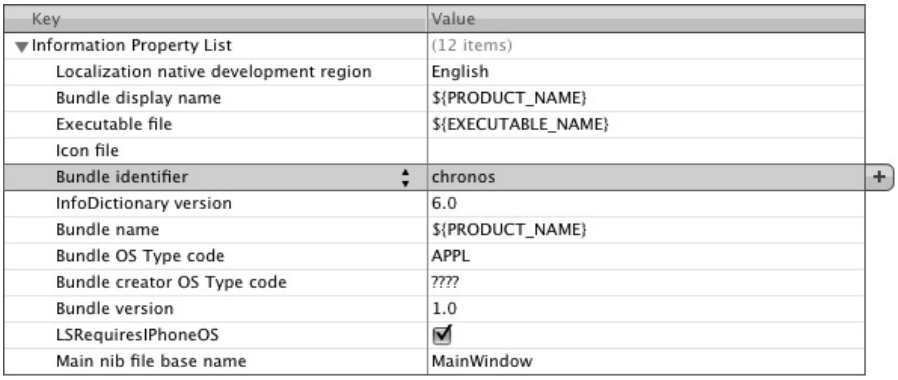


Bild 8.15: Bundle Identifier in `Info.plist`

Wurden alle Schritte erfolgreich durchgeführt, kann das *Active SDK* auf den Eintrag *iPhone Device* gesetzt und die App auf dem Gerät gestartet werden. Natürlich muss dazu das iPhone mit dem Mac verbunden sein.

Der in diesem Unterkapitel besprochene Prozess ist leider relativ komplex und fehleranfällig. Im Internet sind zu dem Thema unglaublich viele Einträge frustrierter Entwickler zu finden. Leider sind dort auch viele falsche oder inzwischen veraltete »Lösungen« beschrieben. Wendet man eine solche an, kann das die Sache weiter verkomplizieren, weil die Ausgangssituation verändert wird. Sollte etwas nicht klappen, so lesen Sie das Unterkapitel zunächst erneut und schauen sich die Abbildungen genau an. Führt das nicht zum Ziel, sollten vor einer allgemeinen Internetrecherche zunächst die Apple-Dokumentation und das iPhone Developer-Forum zu Rate gezogen werden. Auch das Beenden und Neustarten von Xcode und iPhone kann manchmal Wunder bewirken.

8.5 Was die App im Innersten zusammenhält

Hat alles geklappt, dann startet die Anwendung nun auch auf dem Gerät und präsentiert ihr bis jetzt eher spartanisches User Interface. Um erste Anpassungen vornehmen zu können, ist es wichtig zu wissen, wie die einzelnen Bestandteile zusammenspielen. Welche Files werden wann geladen und welche Objekte erzeugt? Zum besseren Nachvollziehen der folgenden Besprechung sollten Sie möglichst die entsprechenden Stellen in Xcode und Interface Builder aufsuchen oder aber zumindest das komplexe Geschehen mithilfe der folgenden Abbildung verfolgen.

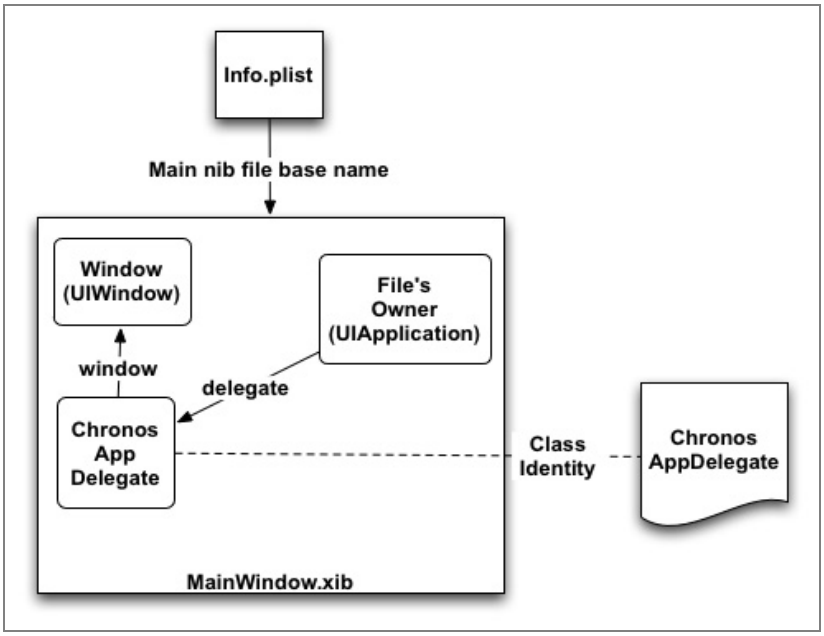


Bild 8.16: Zusammenhänge im Projekt

Beim Start der Anwendung ruft das System die `main()`-Funktion auf. Wie wir schon gesehen haben, wird in `main()` die Funktion `UIApplicationMain` ausgeführt. Sie bekommt vier Parameter übergeben:

```
int retVal = UIApplicationMain(argc, argv, nil, nil);
```

Ein Doppelklick bei gedrückter Apfeltaste auf *UIApplicationMain* zeigt die Definition der Funktion in `UIApplication.h` an. Hier erhält man weitere Informationen über die Parameter:

```
int UIApplicationMain(int argc, char *argv[], NSString *principalClassName,
NSString *delegateClassName);
```

Bei den ersten beiden Werten handelt es sich lediglich um die Weitergabe der Kommandozeilen-Argumente, mit denen `main()` aufgerufen wurde. Das ist eher uninteressant.

Spannender sind die Parameter drei und vier. Sie stehen für die Hauptklasse und das `ApplicationDelegate` der Anwendung. In der durch das Template generierten `main()`-Funktion sind beide `nil`. Ist das der Fall, werden stattdessen Default-Werte genutzt.

Für die Hauptklasse ist der Default-Wert die Klasse `UIApplication`. Eine Singleton-Instanz dieser Klasse wird erzeugt und ist über den folgenden Aufruf stets verfügbar:

```
[UIApplication sharedApplication];
```

Das `ApplicationDelegate` wird, wenn der entsprechende Parameter nicht angegeben ist, im Haupt-Nib-File gesucht. Wie wir schon gesehen haben enthält die Property-Datei `Info.plist` den Namen dieser Datei. `MainWindow.xib` wird von der Instanz des

`UIApplication`-Objekts geladen, wodurch sich dieses zum *File's Owner* des Nib-Files qualifiziert.

Im Nib-File ist die gesuchte Instanz des `ApplicationDelegate` enthalten. Sie ist als *delegate*-Property des *File's Owner*, also des Application-Objekts gesetzt:

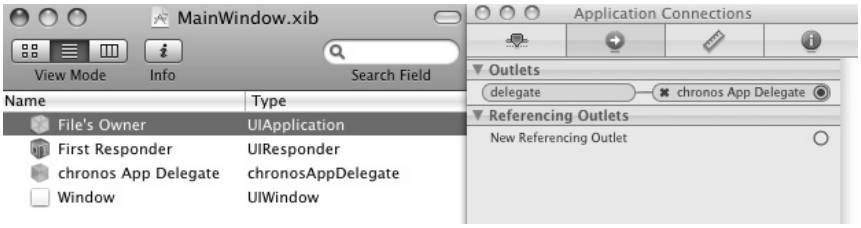


Bild 8.17: Delegate von UIApplication

Es handelt sich hier um keine generische Klasse, sondern um eine Instanz der Klasse `ChronosAppDelegate`, die in Xcode als Sourcecode vorliegt. Der Typ der Klasse ist als Wert für die Property *Class Identity* im Identity-Inspektor angegeben:

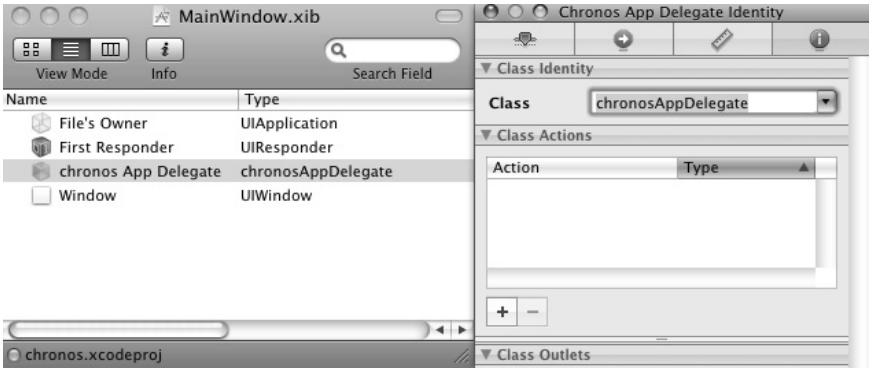


Bild 8.18: Class Identity des App Delegate

Über das `UIApplication`-Objekt kann überall im Code auch relativ einfach auf das AppDelegate zugegriffen werden:

```
ChronosAppDelegate *appDelegate = (ChronosAppDelegate *)[[UIApplication
sharedApplication] delegate];
```

Das AppDelegate kennt über Outlets die ebenfalls im Nib-File vorhandene Instanz von `UIWindow`:

```
@property (nonatomic, retain) IBOutlet UIWindow *window;
```

Bei `window` handelt es sich wirklich um eine Instanz der Klasse `UIWindow` und nicht etwa um eine Implementierung einer in Xcode definierten Subklasse.

Nach dem Laden des Nib-Files ruft das Application-Objekt die Lebenszyklusmethoden seines Delegates auf. Lediglich die Methode `applicationDidFinishLaunching:` ist in der Klasse `ChronosAddDelegate` implementiert:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    // Override point for customization after app launch
    [window makeKeyAndVisible];
}
```

Mit `makeKeyAndVisible` wird das Window angezeigt. Alle iPhone Apps haben nur ein Window, aber oft mehrere Views. Die App befindet sich nun in der Haupt-Eventschleife, wo sie auf Benutzereingaben oder äußere Events wartet.

9 Das Datenmodell: Core Data

Um Daten lokal abspeichern zu können, ist auf dem iPhone eine *SQLite*-Datenbank [URL-SQLITE] verfügbar. Seit der Version 3.0 des iPhone SDK kann glücklicherweise das Persistenz-Framework *Core Data* zum Zugriff auf die Datenbank benutzt werden. Im Vergleich zur recht rudimentären *SQLite*-API steht damit ein mächtiges Werkzeug zur Erstellung von Datenbankzugriffen zur Verfügung. *Core Data* kümmert sich nicht nur um das Abspeichern des Objektgraphen, sondern stellt zusätzlich auch die benötigten Datenobjekte (in unserem Beispiel `Kunde`, `Projekt`, `Leistung` und `Zeit`) zur Verfügung. Das Implementieren dieser Klassen und eine Menge anderer Code entfallen dadurch.

9.1 Modell

Um *Core Data* nutzen zu können, muss zunächst ein Modell der Datenobjekte mit ihren Attributen und den Beziehungen zwischen ihnen angelegt werden. Das Modell wird in einer Datei mit der Endung `.xcdatamodel` gespeichert. Man erhält die Datei entweder, wie im letzten Kapitel demonstriert, durch Verwenden einer Projektvorlage mit *Core Data*-Unterstützung oder aber bei bereits bestehenden Applikationen durch Zufügen eines Data Models via *File > New File…* und Auswahl von *Data Model* aus dem Tab *Resource* im Dialog. Die erstgenannte Variante hat den Vorteil, dass der Code zum Zugriff auf das Modell mitgeneriert wird.

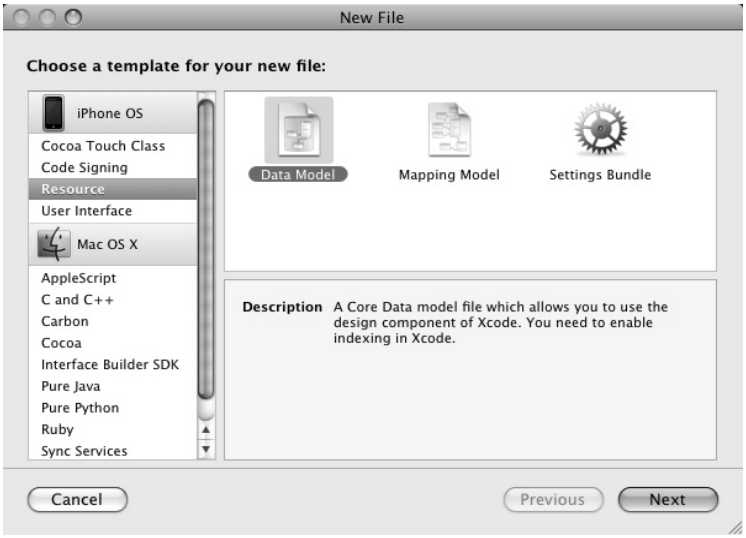


Bild 9.1: Ein Datenmodell zum Projekt hinzufügen

Ein Klick auf die Datei `chronos.xcdatamodel` öffnet den Editor zum Erstellen und Bearbeiten des Datenmodells.

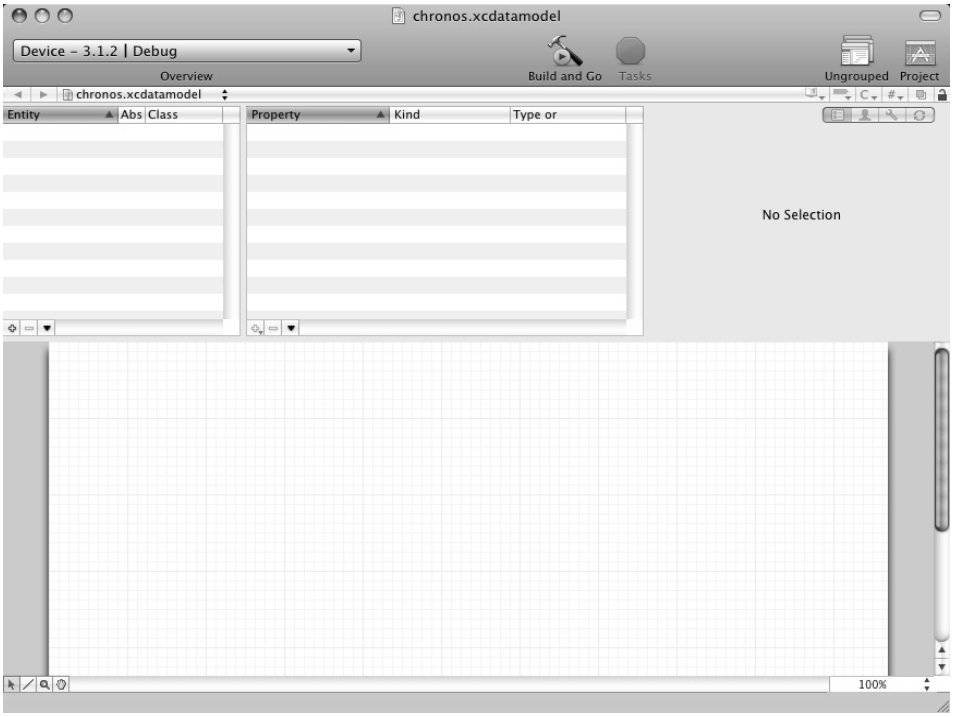


Bild 9.2: Datenmodell-Editor

Der Editor ist in vier Bereiche aufgeteilt. In der oberen Fensterhälfte finden sich die Views für *Entities*, *Properties* (Attribute oder Beziehungen) und *Details*. Im unteren Bereich können die erzeugten Entitäten zur besseren Übersicht visuell angeordnet werden.

9.2 Entitäten

Entitäten (*Entities*) sind Datenobjekte, wie zum Beispiel `Projekt` oder `Kunde`. Jeder Entität entspricht eine Tabelle in der Datenbank.

Eine neue Entität kann im Menü über *Design* > *Data Model* > *Add Entity* angelegt werden (alternativ darf auch das Pluszeichen im Entity-View links oben benutzt werden). Durch einen Doppelklick auf den Namen der Entität kann dieser geändert werden.

9.3 Attribute

Attribute in Entitäten entsprechen den Spalten in Datenbanktabellen. Hier werden Werte der Entität, wie zum Beispiel der Name eines Projekts, gespeichert.

Analog zu Entitäten können Attribute bei markiertem Entity über das Menü via *Design* > *Data Model* > *Add Attribute* oder aber über das Pluszeichen im Property-View zugefügt werden.

Im Detail-Bereich ganz rechts lassen sich wichtige Eigenschaften des Attributs wie etwa das Optional-Flag, der Datentyp oder der Default-Wert anpassen:

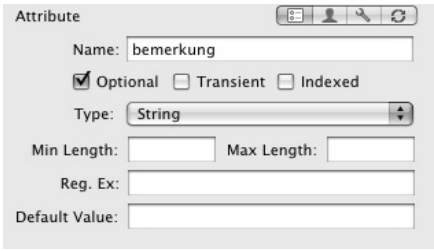


Bild 9.3: Entity-Attribute bearbeiten

Das Aussehen des Bereichs variiert mit dem ausgewählten Datentyp.

9.4 Beziehungen

Für eine Beziehung (*Relationship*) wird im Gegensatz zu Attributen kein Typ oder Default-Wert angegeben. Stattdessen definiert man unter anderem eine Ziel-Entity, evtl. eine *Inverse Relationship* und die Kardinalität.

Die Ziel-Entity gibt an, mit welcher Art von Entity die bearbeitete Entity verknüpft ist. Klassisches Beispiel ist eine Firma mit Mitarbeitern. Möchte man diesen Sachverhalt als Modell darstellen, muss eine Beziehung von Firma zu Mitarbeiter angelegt werden. Da eine Firma im Allgemeinen mehrere Mitarbeiter hat, wird die Beziehung von der Kardinalität *To-Many* sein. Umgekehrt kann natürlich auch eine Beziehung von Mitarbeiter zu Firma modelliert werden. Da ein Mitarbeiter in der Regel nur bei einer Firma arbeitet, wird diese *To-One* sein. In diesem Fall wird die *To-Many*-Checkbox im Detail-View also nicht angehakt.

Die zweite Beziehung kann als *Inverse Relationship* der ersten Beziehung definiert werden. Im *Core Data Programming Guide* von Apple wird dem Entwickler nahegelegt, zur Wahrung der Konsistenz des Objektgraphen die Beziehungen stets in beide Richtungen zu modellieren.

Bild 9.4: Bearbeiten von Entity-Beziehungen

Eine weitere wichtige Option beim Anlegen von Beziehungen ist die *Delete Rule*. Die Regel beschreibt, wie mit den in Beziehung stehenden Entitäten (also etwa den Mitarbeitern einer Firma) verfahren wird, wenn die Source-Entity (also die Firma) gelöscht wird. Mit der Auswahl von *Cascade* würden die Mitarbeiter gleich mit gelöscht, *Nullify* lässt die Mitarbeiter bestehen, allerdings ohne Bezug zur Firma. *Deny* verhindert das Löschen der Firma, solange noch Mitarbeiter existieren.

9.5 Erstellen des Chronos-Datenmodells

Für *Chronos* werden die Entitäten Kunde, Projekt, Leistung und Zeit mit folgenden Attributen und Beziehungen benötigt:

9.5.1 Kunde

Beim Anlegen eines Kunden in der App muss ein Name und kann eine Bemerkung angegeben werden. Es ergeben sich damit folgende Attribute für die Entity Kunde:

Attribut	*Typ*	*Optional*
`name`	String	Nein
`bemerkung`	String	Ja

Für einen Kunden können mehrere Projekte angelegt werden. Die Beziehung ist also vom Typ *To-Many*. Es kann natürlich Kunden geben, für die noch kein Projekt angelegt wurde. Die Beziehung ist folglich optional. Als *Delete Rule* wird *Nullify* vergeben. Das Löschen eines Kunden löscht also nicht die zugehörigen Projekte mit (wie es bei *Cascade* der Fall wäre) und wird auch nicht verhindert, wenn noch zugeordnete Projekte existieren (wie es bei *Deny* der Fall wäre). Stattdessen wird der Kunde gelöscht und die Projekte bleiben ohne Bezug zum Kunden stehen. Da Projekte ohne zugeordneten Kunden in der App erlaubt sein sollen, ergibt das Sinn:

Beziehung	*Destination*	*Optional*	*To-Many*	*Delete Rule*
`projekt`	Projekt	Ja	Ja	*Nullify*

Man beachte bitte bei den Beziehungen des Projekts weiter unten die inverse Beziehung zum Kunden.

Die Überlegungen für das Modellieren der restlichen Entitäten erfolgen analog und werden deshalb nur noch in Tabellenform wiedergegeben.

9.5.2 Projekt

Attribut	*Typ*	*Optional*
`name`	String	Nein
`bemerkung`	String	Ja
`budget`	Decimal	Ja
`budgetTyp`	String	Ja

Beziehung	*Destination*	*Optional*	*To-Many*	*Delete Rule*
`kunde`	Kunde	Ja	Nein	*Nullify*
`zeit`	Zeit	Ja	Ja	*Nullify*

9.5.3 Leistung

Attribut	*Typ*	*Optional*
`name`	String	Nein
`stundensatz`	Decimal	Ja
`bemerkung`	String	Ja
`fakturierbar`	Boolean	Nein

Beziehung	*Destination*	*Optional*	To-Many	*Delete Rule*
`zeit`	Zeit	Ja	Ja	*Nullify*

9.5.4 Zeit

Property	*Typ*	*Optional*
`datum`	Date	Nein
`bemerkung`	String	Ja
`dauer`	Decimal	Nein

Beziehung	*Destination*	*Optional*	*To-Many*	*Delete Rule*
`projekt`	Projekt	Ja	Nein	*Nullify*
`leistung`	Leistung	Ja	Nein	*Nullify*

Ergänzend zu den Angaben in den Tabellen muss für jede Beziehung die passende *Inverse Relationship* ausgewählt werden.

Nach Anlegen der Entitäten mit ihren Attributen und Beziehungen ergibt sich folgendes Bild:

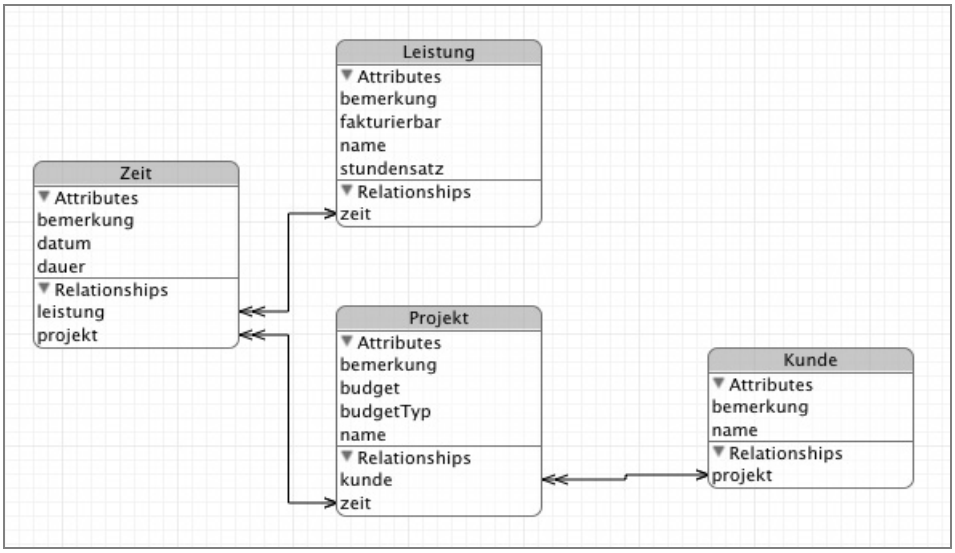


Bild 9.5: Fertiges Datenmodell

Beziehungen mit Pfeilen auf beiden Seiten symbolisieren das Vorliegen einer *Inverse Relationship*. Die Doppelpfeile deuten auf eine *To-Many*-Beziehung hin.

9.6 Die Zugriffsschicht: der Core-Data-Stack

Durch Wahl einer Projektvorlage mit *Core Data Model* wurde für das Projekt neben dem leeren Datenmodell auch die Zugriffsschicht für die Persistenz, der *Core Data Stack*, erzeugt (siehe folgende Abbildung). Er umfasst die Klassen `NSManagedObjectContext`, `NSManagedObjectModel`, und `NSPersistentCoordinator`.

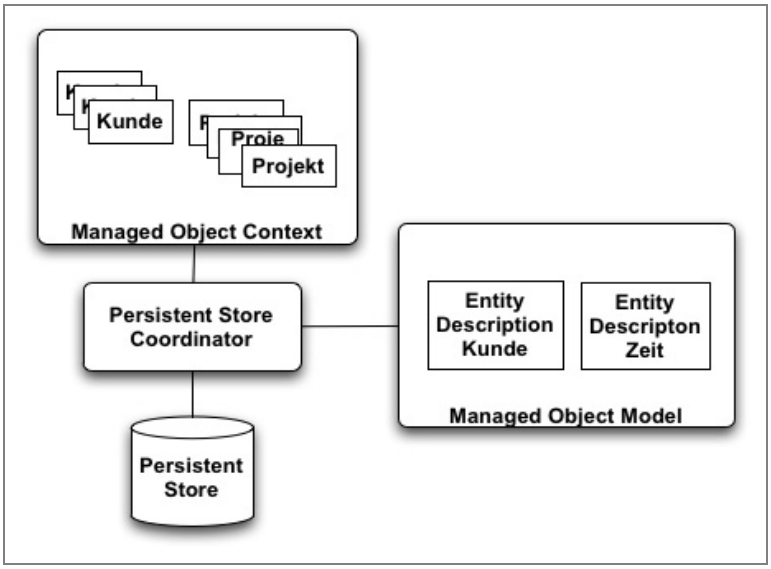


Bild 9.6: Core-Data-Stack

Der gesamte Zugriffscode findet sich bereits in der Klasse `ChronosAppDelegate`, sodass für die Zeiterfassung keine weiteren Arbeiten notwendig sind.

Zum Verständnis werden die Klassen hier kurz diskutiert. Sollten Sie gerade erst mit Objective-C beginnen, stellt das Verstehen des generierten Codes womöglich eine ordentliche Herausforderung dar. Nutzen Sie die Gelegenheit und sehen Sie sich die Methoden genau an. Eine der besten Möglichkeiten, eine Programmiersprache zu erlernen, besteht darin, existierenden Code zu untersuchen.

9.6.1 Managed Object Model

`NSManagedObjectModel` repräsentiert das Datenmodell und ist somit im Prinzip eine kompilierte Version der vorhandenen `.xcdatamodel`-Dateien. Darin enthalten sind zum Beispiel die Beschreibungen aller angelegten Entitäten in Form von Instanzen der Klasse `NSEntityDescription`.

Zur Erzeugung einer Instanz von `NSManagedObjectModel` kann die Klassenmethode `mergedModelFromBundles:` genutzt werden. Sie erzeugt ein Modell durch Zusammenfügen aller `.xcdatamodel`-Dateien, die im angegebenen Bundle gefunden werden. Wird statt des Bundle `nil` angegeben, wird das `MainBundle` verwendet.

Der folgende Code zeigt die von der Projektvorlage erzeugte Methode zur Erzeugung der Instanz von `managedObjectModel`.

Das Modell wird in der Instanzvariablen vorgehalten, damit der Merge-Prozess nicht bei jedem Zugriff erneut durchgeführt werden muss:

```
- (NSManagedObjectModel *)managedObjectModel {
   if (managedObjectModel != nil) {
     return managedObjectModel;
   }
   managedObjectModel = [[NSManagedObjectModel mergedModelFromBundles:nil]
     retain];
   return managedObjectModel;
}
```

Soll kein zusammengeführtes Modell aus mehreren `.xcdatamodel`-Dateien erzeugt werden, kann auch gezielt über die Methode `initWithContentsOfURL:` eine bestimmte Datei zur Erzeugung der Instanz bestimmt werden. Im Beispielprojekt macht das keinen Unterschied, weil ohnehin nur eine Modelldatei existiert.

9.6.2 NSPersistentStoreCoordinator

Ganz unten im Zugriffsstapel befindet sich der *Persistent Store Coordinator*. Er verbindet Modell und Datenbank, ist also zuständig für das Speichern der Daten in der Datenbank.

Zur Erzeugung einer Instanz des Koordinators wird in der Methode `initWithManagedObjectModel` das zuvor erzeugte *Managed Object Model* übergeben. Anschließend wird mittels `addPersistentStoreWithType:configuration:URL:options:error:` ein *Persistent Store* vom Typ `NSSQLiteStoreType` zugefügt. Der *Persistent Store* ist der Speicher für die Daten aus dem Modell. Beim Hinzufügen des Store wird der Pfad zur *SQLite*-Datei übergeben:

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (persistentStoreCoordinator != nil) {
        return persistentStoreCoordinator;
    }
    NSURL *storeUrl = [NSURL fileURLWithPath: [[self
     applicationDocumentsDirectory] stringByAppendingPathComponent:
     @"chronos.sqlite"]];

   NSError *error;
    persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
     initWithManagedObjectModel: [self managedObjectModel]];
    if (![persistentStoreCoordinator
addPersistentStoreWithType:NSSQLiteStoreType
     configuration:nil URL:storeUrl options:nil error:&error]) {
        // Handle error
    }

    return persistentStoreCoordinator;
}
```

Es sei an dieser Stelle angemerkt, dass Core Data nicht zwangsläufig eine Datenbank als Repository nutzen muss. Neben zahlreichen anderen Speichervarianten ist zum Beispiel auch das Aufbewahren des Objektgraphen im Speicher möglich.

9.6.3 NSManagedObjectContext

`NSManagedObjectContext` bildet die oberste Schicht im Zugriffs-Stack und stellt die Verbindung zwischen der Persistenzschicht und dem Anwendungscode dar. Immer, wenn Daten gelesen oder geschrieben werden sollen, wird die Klasse benutzt. Eine Instanz von `NSManagedObjectContext` kann mit `alloc/init` erzeugt werden und bekommt den *Persistent Store Coordinator* gesetzt:

```
- (NSManagedObjectContext *) managedObjectContext {

    if (managedObjectContext != nil) {
        return managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator = [self
persistentStoreCoordinator];
    if (coordinator != nil) {
        managedObjectContext = [[NSManagedObjectContext alloc] init];
        [managedObjectContext setPersistentStoreCoordinator: coordinator];
    }
    return managedObjectContext;
}
```

9.7 Verwenden des Stack

Der Stack-Code kann wie erwähnt durch Auswahl der Core-Data-Option in der Projektvorlage generiert werden. Auch wenn Ihnen die Interna nicht hundertprozentig klar sind, können Sie ihn vorerst einfach nutzen. Anders ist das beim Code, der den Stack verwendet, also den eigenen Abfragen an die Datenbank. Die dafür benötigten Klassen werden in diesem Abschnitt vorgestellt. Die Verwendung wird in den danach folgenden Kapiteln demonstriert.

9.7.1 Fetch Request

Die Klasse `NSFetchRequest` wird benutzt, um gespeicherte Objekte aus dem Core-Data-Repository zu holen. Dafür muss natürlich mindestens angegeben werden, um welchen Typ von Objekten es sich handelt. Für diese Aufgabe wird die Klasse `NSEntityDescription` bzw. ihre Klassenmethode `entityForName: inManagedObjectContext:` herangezogen. Wenn klar ist, was gesucht werden soll, wird der Request mit `executeFetchRequest:error:` ausgeführt. Die Methode findet sich in der Klasse `NSManagedObjectContext`, die wie erwähnt die Verbindung zwischen der

Persistenzschicht und dem Anwendungscode darstellt. Die Ergebnisse der Suche werden in einem `NSArray` erhalten:

```
NSFetchRequest *request = [[NSFetchRequest alloc]init];
[request setEntity:[NSEntityDescription entityForName:@"Kunde"
  inManagedObjectContext:managedObjectContext]];
NSError *error;
NSArray *results = [managedObjectContext
  executeFetchRequest:request error:&error];
if (error)
{
  // handle error
}
```

9.7.2 Filtern: NSPredicate

Oft werden nicht alle Instanzen eines Entity-Typs (also nicht alle Zeilen einer Tabelle) benötigt. Das Ergebnis muss gefiltert werden, wozu die Klasse `NSPredicate` dient. Mit ihrer Hilfe können Filterkriterien ähnlich den `Where`-Klauseln in SQL-Statements definiert werden.

In folgendem Beispiel werden nur die fakturierbaren Leistungen geladen:

```
NSUInteger fakturierbar = 1;
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:[NSEntityDescription entityForName:@"Leistung"
   inManagedObjectContext:managedObjectContext]];
NSPredicate *predicate = nil;
predicate = [NSPredicate predicateWithFormat:@"fakturierbar = =%i" ,
fakturierbar];
[request setPredicate:predicate];
```

Statische *Fetch Requests* können nicht nur im Code, sondern alternativ auch im Datenmodell als benannte Queries hinterlegt werden. Die zugehörige Funktion findet sich im Menü unter *Design > Data Model > Add Fetch Request*. Damit der Menüpunkt aktiviert ist, muss zuvor die Entität ausgewählt werden, für die ein Request erstellt werden soll.

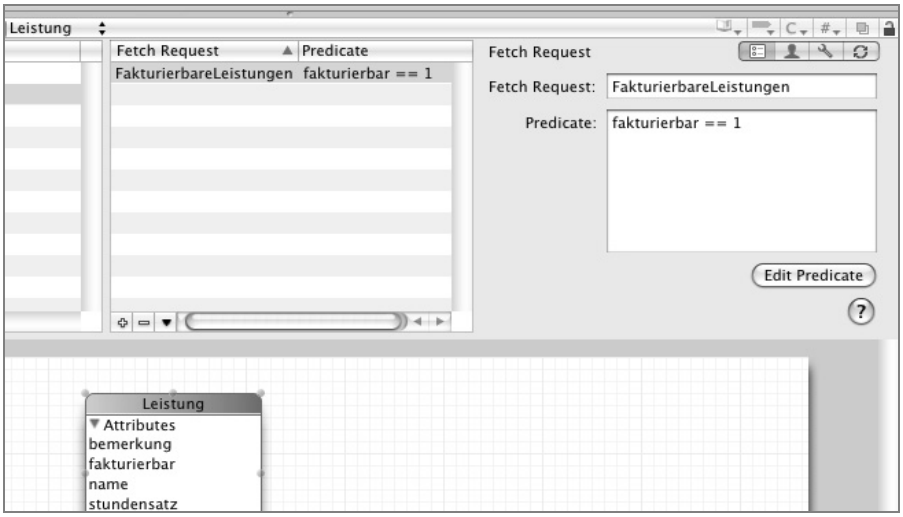


Bild 9.7: Stored Fetch Request

Der Zugriff auf den im Datenmodell hinterlegten Request erfolgt über den Namen:

```
NSFetchRequest *request =
[managedObjectModelfetchRequestTemplateForName:@"FakturierbareLeistungen"];
NSError *error = nil;
NSArray *result = [managedObjectContext executeFetchRequest:request
error:&error];
if (error) {
// handle error
}
```

9.7.3 Sortieren: NSSortDescriptor

Neben dem Filtern ist das Sortieren der Daten eine gängige Anforderung. Auch hierfür gibt es eine Lösung, den `NSSortDescriptor`. Eine Instanz wird mit dem Namen des Attributs, nach dem sortiert werden soll, und der gewünschten Richtung erzeugt. Dem *Fetch Request* wird dann ein Array mit allen gewünschten Sortierkriterien gesetzt:

```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
initWithKey:@"name"
    ascending:YES];
NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor,
nil];
[fetchRequest setSortDescriptors:sortDescriptors];
```

9.7.4 Managed Object

Das Ergebnis eines *Fetch Requests* ist ein `NSArray` mit Objekten. Alle im Datenmodell definierten Entitäten können praktischerweise verwendet werden, ohne dass entsprechende Subklassen (wie `Kunde` oder `Projekt`) erzeugt werden müssen. Sie werden dann im Ergebnis-Array durch Instanzen der Klasse `NSManagedObject` repräsentiert.

Die Verwendung dieser generischen Klasse bringt ein Problem mit sich. Existiert keine konkrete Subklasse, so existiert natürlich auch kein Accessor, wie zum Beispiel `getBemerkung`. Dank einer *Key Value Coding (KVC)* [AKVC10] genannten Technologie sind trotzdem Zugriffe auf die im Data Model definierten Attribute und Beziehungen einer Entität möglich. Hierbei werden die Properties eines Objekts nicht über `get-` und `set`-Methoden gelesen oder gesetzt, sondern mit Hilfe eines Strings, der als Key für die entsprechende Property fungiert. Der KVC-Zugriff auf das Attribut `bemerkung` sieht folgendermaßen aus:

```
NSString *bemerkung = [aManagedObject valueForKey:@"bemerkung"];
```

Beim Zugriff auf eine Beziehung vom Typ *To-Many* erhält man ein Set von Objekten zurück:

```
NSSet *mitarbeiter = [aManagedObject valueForKey:@"mitarbeiter"];
```

Auch das Setzen von Werten ist mit KVC möglich:

```
[aManagedObject setValue:@"Eine Bemerkung" forKey:@"bemerkung"];
```

Natürlich können bei Bedarf auch Subklassen von gemanagten Objekten erzeugt werden. Für *Chronos* werden die vier Klassen `Kunde`, `Projekt`, `Leistung` und `Zeit` angelegt. Die jeweilige Modell-Klasse erbt dann von `NSManagedObject` und führt die im Modell definierten Attribute als Properties auf:

```
@interface Kunde : NSManagedObject  {

// Keine Instanzvariablen!
}

@property (nonatomic, retain) NSString *name;
@property (nonatomic, retain) NSString *bemerkung;

@end
```

Wie dem Interface entnommen werden kann, erfolgt die Deklaration der Properties, ohne dass zugehörige Instanzvariablen vorhanden sind. Die Instanzvariablen sind ja bereits im Datenmodell als Attribute beschrieben. Damit der Compiler zufrieden ist, muss ihm in der Implementierungsdatei durch `@dynamic` versichert werden, dass die im Moment noch fehlenden Variablen dynamisch hinzugefügt werden:

```
#import "Kunde.h"

@implementation Kunde
```

```
@dynamic name;
@dynamic bemerkung;

@end
```

Ist die Subklasse angelegt, dann kann in gewohnter Weise auf die Properties zugegriffen werden:

```
NSString *bemerkung = kunde.bemerkung;
```

Sollten `get`- und `set`-Methode in einem von `NSManagedObject` abgeleiteten Objekt existieren, würden diese übrigens auch beim KVC-Zugriff benutzt werden.

Für die Anforderungen der Zeiterfassung genügen die in diesem Kapitel besprochenen Zusammenhänge. Falls Sie versuchen, das Entstehen der Zeiterfassung mitzuprogrammieren, dann sollten nach diesem Kapitel das gefüllte Datenmodell und die Klassen `Kunde`, `Projekt`, `Leistung` und `Zeit` vorliegen. Wer tiefer in Core Data einsteigen möchte, dem sei das Buch von Marcus Zarra [ZAR09] empfohlen.

10 View Controller

Die enge Beziehung zwischen Views und ihren Controllern als Folge des in Cocoa allgegenwärtigen MVC-Patterns wurde schon an mehreren Stellen besprochen und demonstriert. *View Controller* verbinden Views mit dem Rest der Anwendung und bilden die Brücke zwischen View und Model. Der Controller ist dabei der Teil der Anwendung, in dem der Entwickler wirklich gefordert wird. Views lassen sich per Drag & Drop im Interface Builder erzeugen, das Datenmodell ähnlich einfach mit Core Data. Im Extremfall liegt für beides kein selbst geschriebener Code mehr vor. Beim Controller ist das anders, hier ist noch »richtige« Programmierarbeit in Form von Objective-C-Quellcode gefragt. In diesem Kapitel sollen noch mal einige Grundlagen besprochen werden, mit *Chronos* geht's dann im Kapitel *Navigation Controller* weiter.

10.1 Views & Controller

Jeder View der Anwendung wird von einem *View Controller* dargestellt. Dabei gilt die Regel, dass jeder Controller nur für einen View, also einen Screen mit Inhalt, verantwortlich sein sollte. Der View seinerseits sollte immer den kompletten Screen abdecken, also kein Teil eines Views sein.

Der Controller hat Zugriff auf die darzustellenden Daten (das Model) und teilt dem View mit, welche Daten wie dargestellt werden sollen. Außerdem ist der View Controller für das Ausführen von Aktionen, zum Beispiel nach einem Button-Tap im View, zuständig.

Ein Beispiel für diese quasi klassischen View Controller ist der *Table View Controller* mit seinen Subklassen. Er ist zuständig für die Kommunikation mit dem Table View, stellt die Daten aus dem Model bereit und enthält Methoden, die beispielsweise beim Tap auf eine Zelle aufgerufen werden.

Neben diesen typischen View Controllern nehmen der *Tab Bar Controller* und der *Navigation Controller* eine besondere Stellung ein. Sie sind Container für andere View Controller und managen die Navigation zwischen deren Views.

Wie dem folgenden Klassendiagramm zu entnehmen ist, erben alle *View Controller* von der Klasse `UIViewController`:

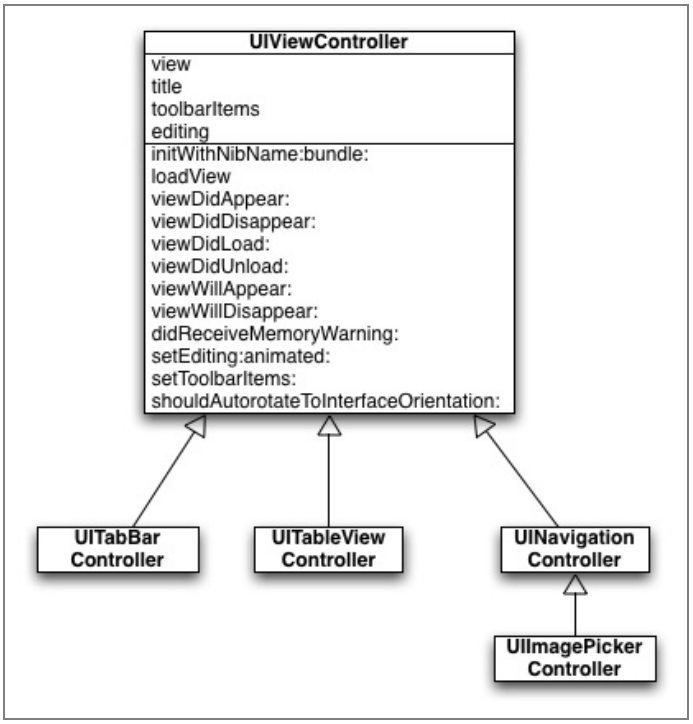


Bild 10.1: View Controller-Vererbungshierarchie

Das Klassendiagramm enthält aus Platzgründen nur eine Auswahl der wichtigsten Attribute und Methoden.

10.2 UIViewController

Die Klasse `UIViewController` stellt die fundamentale Infrastruktur für alle View Controller zur Verfügung. Im Normalfall wird keine Instanz dieser Klasse, sondern die einer Subklasse verwendet.

10.2.1 Erzeugen von View & Controller

Grundsätzlich kann ein View auf zwei verschiedene Arten mit einem View Controller verbunden werden.

Bei der ersten Variante wird der View im Interface Builder konstruiert. Aus Gründen der Übersichtlichkeit ist es sinnvoll, ein eigenes Nib-File für jeden View anzulegen. In Xcode kann dies über *File > New File... > User Interface > View Xib* geschehen:

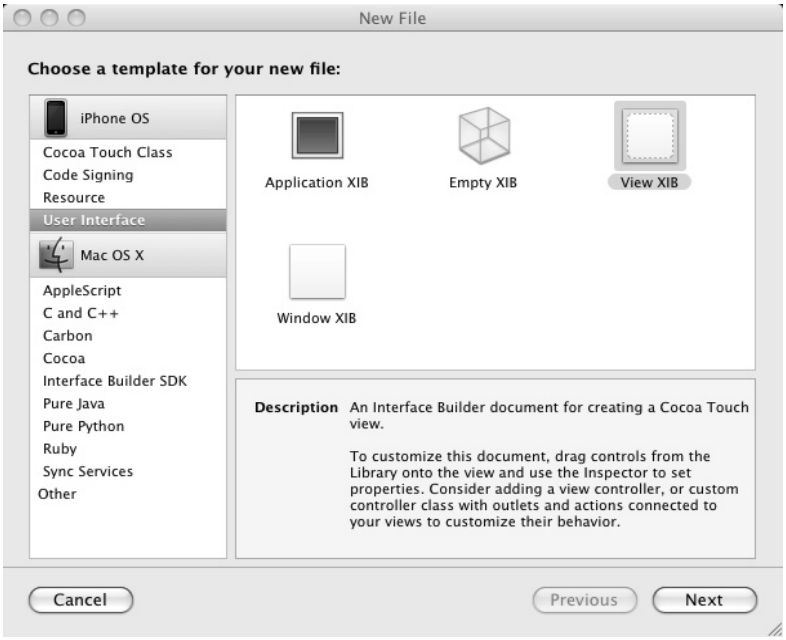


Bild 10.2: Anlegen eines View-Xib

Im Xcode-Projekt wird außerdem eine Subklasse von `UIViewController` erzeugt und die Klasse (Class Identity) des *File's Owner* im Nib-File auf diese Subklasse gesetzt. Letzteres geschieht, wie schon erläutert, im Identity-Inspektor.

Das `view`-Outlet der View-Controller-Subklasse wird im Connections-Inspektor mit dem erzeugten View im Nib-File verbunden. Die folgende Abbildung verdeutlicht die Zusammenhänge:

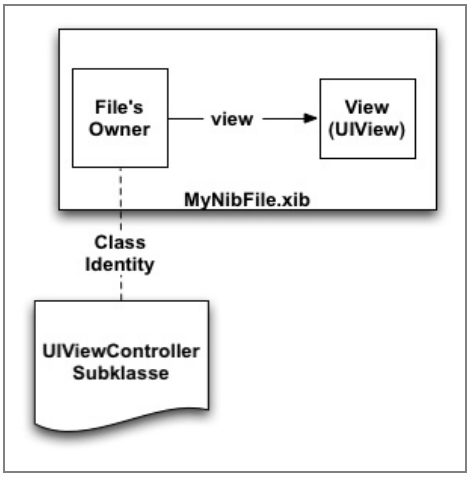


Bild 10.3: Zusammenspiel von View Controller und Nib-File

Ein View Controller, dessen View im Interface Builder erzeugt wurde, wird im Code mit der Methode `initWithNibName:bundle:` initialisiert. Als erster Parameter wird dabei der Name des Nib-Files (ohne die Endung `.xib`) übergeben. Beispiel:

```
MyViewController *myViewController =
[[MyViewController alloc] initWithNibName:@"MyNibFile"
   bundle:nil];
```

Der `bundle`-Parameter betrifft die Internationalisierung und kann als `nil` übergeben werden. Vom *View Controller* kann über die `view`-Property der im Nib-File definierte View erhalten werden und zum Beispiel dem Window der App zugefügt werden:

```
[window addSubview:myViewController.view];
```

Das Nib-File und der darin enthaltene View werden nicht direkt beim Ausführen der Methode geladen, sondern bei der ersten Benutzung. Man spricht hier von *Lazy Loading*. Trotz dieses Kniffs ist das Laden und Auswerten des Nib-Files ein relativ zeitaufwendiger Vorgang. Um die Erstellung der Views ein wenig zu beschleunigen, empfahl Apple zumindest bei Erscheinen der ersten iPhone-Generation als Alternative das Erzeugen des Views im Code.

Zu diesem Zweck kann die Methode `loadView` in der Subklasse von `UIViewController` überschrieben werden. Sie wird aufgerufen, wenn der View benötigt wird und bisher nicht erstellt wurde. Diese Vorgehensweise ist im Vergleich zur Arbeit mit dem Interface Builder natürlich deutlich unkomfortabler, wie folgendes Beispiel zeigt:

```
- (void)loadView
{
   [super loadView];
   CGRect frame = CGRectMake(10, 10, 300, 300);
   UILabel  *label = [[UILabel alloc] initWithFrame:frame];
   label.textAlignment = UITextAlignmentCenter;
   label.font = [UIFont fontWithName:@"Verdana" size:48];
   label.text = @"Hello World";
   [self.view addSubview:label];
   [label release];
}
```

Im Code wird ein Label, basierend auf den Dimensionen eines `CGRect`-Objekts, erstellt. Anschließend wird das Label zentriert, bekommt Schriftart und -größe zugewiesen und den Text »Hello World« gesetzt.

Der View Controller, der diesen Code enthält, wird mit `alloc/init` und nicht mehr mit `initWithNibName:bundle:` erzeugt (es muss ja kein Nib-File mehr geladen werden).

Was für Java-Swing-Entwickler viele Jahre harte Realität war, ist für den an Komfort gewöhnten AppKit-Mac-Entwickler nur schwer hinnehmbar. Inzwischen sollte sich dank schnellerer Prozessoren das Problem aber auch relativiert haben, sodass nur noch in speziellen Einzelfällen (wenn die Performance eben nicht zufriedenstellend ist) auf diesen steinigen Weg ausgewichen werden muss.

10.2.2 Wichtige Methoden

Die Superklasse `UIViewController` verfügt über eine Reihe von Methoden, die bei bestimmten Ereignissen aufgerufen werden. Durch Überschreiben der Methoden in der Subklasse lassen sich so Initialisierungen oder Aufräumarbeiten durchführen. Im Einzelnen sind dies die folgenden Methoden:

viewDidLoad:

`viewDidLoad` wird einmalig aufgerufen, wenn der View geladen wurde, also nach `loadView` oder dem Setzen des Views im Fall eines Xib-Files. Hier lassen sich einmalige Initialisierungen wie das Definieren des Titels oder das Hinzufügen von Items zu einer Toolbar durchführen.

viewDidUnload:

Diese Methode wird aufgerufen, wenn der View freigegeben und auf `nil` gesetzt wurde.

viewWillAppear:

Bei der Verwendung von *Table Views* ist die Methode eine geeignete Stelle, um bei geändertem Datenmodell die Tabelle neu zu laden. Sie wird vor jedem Anzeigen des Views aufgerufen.

viewDidAppear:

`viewDidAppear` wird nach jedem Anzeigen des Views aufgerufen.

viewWillDisappear:

Diese Methode wird vor jedem Verschwinden des Views aufgerufen.

viewDidDisappear:

`viewDidDisappear` wird nach jedem Verschwinden des Views aufgerufen.

didReceiveMemoryWarning:

Das Laufzeitsystem gestattet jeder App, eine bestimmte Speichermenge (abhängig von der zugrunde liegenden Hardware) zu benutzen. Wird der Speicher knapp, ruft das System vor einem Out-of-Memory-Error zunächst die Methode `didReceiveMemoryWarning` des View Controllers auf. Durch Überschreiben der Methode in der Subklasse kann so durch Freigeben nicht mehr benötigter Ressourcen der Out-of-Memory-Error womöglich vermieden werden.

```
- (void)didReceiveMemoryWarning {
   // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];
```

```
    // Release any cached data, images, etc that aren't in use.
}
```

Wenn man die Methode überschreibt, sollte die Implementierung auf jeden Fall ausprobiert werden. Dabei hilft der Simulator. Über das Menü kann via *Hardware > Speicherwarnhinweis simulieren* die Speicherknappheit vorgetäuscht werden.

shouldAutorotateToInterfaceOrientation:

Eines der iPhone-Features, das neue Nutzer sofort begeistert, ist das animierte Rotieren des User Interfaces beim Drehen des Geräts um 90°. Nicht in allen Anwendungen ist das unbedingt sinnvoll. Schön ist, wenn der Wechsel von der Porträt- in die Landscape-Darstellung einen Mehrwert für den Nutzer bringt. Besonders gut ist die Umsetzung in der iPod-App gelungen, wo zwei gänzlich verschiedene Views für die Modi verwendet werden.

Ob der View das Drehen unterstützen soll, kann in der Methode `shouldAutorotateToInterfaceOrientation:` angegeben werden. Wird hier `YES` zurückgegeben, werden beide Ansichten unterstützt. Im Falle eines `NO` funktioniert lediglich die Porträt-Ansicht.

```
-(BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}
```

Auch hier bietet der Simulator die Möglichkeit, die Auswirkungen der Implementierung auszuprobieren. Im Menü finden sich unter *Hardware* die Punkte *Links drehen* und *Rechts drehen.*

Die Basisklasse `UIViewController` stellt also einige hilfreiche Methoden zu den Themen Lebenszyklus-, Memory- und Orientierungs-Management zur Verfügung. Über diese ererbten und überschreibbaren Methoden darf aber eine wichtige Aufgabe der erzeugten *View Controller* nicht vergessen werden. Eine abgeleitete Subklasse ist immer der bevorzugte Kandidat, wenn eine Datenquelle für die anzuzeigenden View-Daten oder ein Handler für auf dem View ausführbare Aktionen gesucht wird. Oft implementieren die View-Controller-Subklassen dazu Protokolle und dienen als Delegate.

Wie im Klassendiagramm zu sehen ist, hat `UIViewController` drei direkte Subklassen:

- `UINavigationController`
- `UITableViewController`
- `UITabBarController`

Dies sind die zentralen Klassen in der iPhone-OS-Programmierung. Wegen ihrer Wichtigkeit werden sie im Folgenden in eigenen Kapiteln vorgestellt. Dabei bietet sich dann auch wieder die Gelegenheit, ein wenig mit dem Zeiterfassungsprojekt voranzukommen.

11 Navigation Controller

Ebenso wie der Tab Bar Controller kann der *Navigation Controller* mehrere View Controller in einem Container-Objekt verwalten. Somit spielt er eine besondere Rolle beim Zusammenstellen komplexer Anwendungen aus einzelnen Views und ihren Controllern.

Die vorrangige Aufgabe des Navigation Controllers ist – insbesondere in Zusammenarbeit mit dem Table View Controller – die Darstellung hierarchischer Daten.

11.1 Funktionsweise des Navigation Controllers

Der Navigation Controller verwaltet die ihm anvertrauten View Controller in Form eines Stapels. View Controller können oben auf den Stapel gelegt oder von der Stapelspitze weggenommen werden. Ein direkter Zugriff auf Elemente innerhalb des Stapels ist ohne Entfernen der darüber liegenden View Controller nicht möglich. Der View des obersten View Controllers im Stapel wird im Content-Bereich der Anwendung angezeigt.

Die Abbildung verdeutlicht die Funktionsweise.

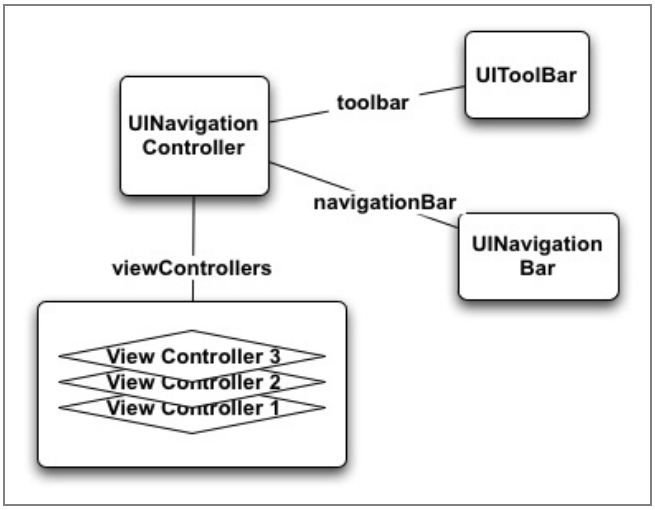


Bild 11.1: Funktionsweise des Navigation Controllers

Oberhalb des Content-Bereichs findet sich die für ein Navigation-Bar-UI typische Navigationsleiste. Auf der linken Seite enthält sie einen Button zum Wechsel in die vorherige Hierarchieebene, also zu dem View Controller, der sich unter dem aktuellen View Controller im Stapel befindet. Der Button erhält als Beschriftung automatisch den Titel

des vorherigen View Controllers. Enthält der Stapel nur noch einen Controller, ist keine Zurück-Navigation mehr möglich, und der Button wird nicht angezeigt. In der Mitte der Navigation Bar wird der Titel des aktuellen, also obersten Controllers angezeigt.

Ein typischer Anwendungsfall für den Navigation Controller ist der Drill-Down in eine tiefer gelegene Hierarchieebene. Zu Beginn des Szenarios ist ein View, zum Beispiel ein Menü in Form einer Tabelle, sichtbar. Dieser View (eigentlich sein Controller) ist zu diesem Zeitpunkt oberstes Element im Stapel. Durch Klicken auf einen Menüpunkt wird in einer mit der Aktion verbundenen Methode ein neuer View, zum Beispiel eine Liste, auf den Stapel gelegt. Der neue View ist jetzt ganz oben auf dem Stapel und damit sichtbar. Das Menü befindet sich im Stapel direkt darunter. Durch Betätigen des Zurück-Buttons auf der Navigationsleiste wird der oberste View vom Stapel genommen und das Menü wird wieder sichtbar.

Bild 11.2: Drill-Down

Die zwei einfachsten Methoden zur Manipulation des Stapels heißen `pushViewController:animated:` und `popViewControllerAnimated:`. Die Methode `popViewControllerAnimated:` zum Entfernen des obersten View Controllers vom Stapel wird automatisch aufgerufen, wenn der Zurück-Button in der *Navigation Bar* geklickt wird. Sie braucht also im Normalfall nicht vom Entwickler aufgerufen zu werden. Die andere Methode, `pushViewController:animated:`, wird durchaus vom Entwickler aufgerufen, es werden später Beispiele dafür folgen.

11.2 Erzeugen eines Navigation View Controllers

Das Erzeugen eines Navigation Controllers im Interface Builder ist keine allzu anspruchsvolle Aufgabe. Zur Demonstration des Vorgehens erweitern wir nun die bisher lediglich aus der generierten Vorlage und dem Core-Data-Datenmodell bestehende Anwendung.

Durch Doppelklick auf die Datei `MainWindow.xib` startet Interface Builder und zeigt das Document Window an. Per Drag & Drop wird nun aus der Library (gegebenenfalls über *Tools* > *Library* öffnen) ein Navigation Controller unter den letzten Eintrag im Document Window gezogen:

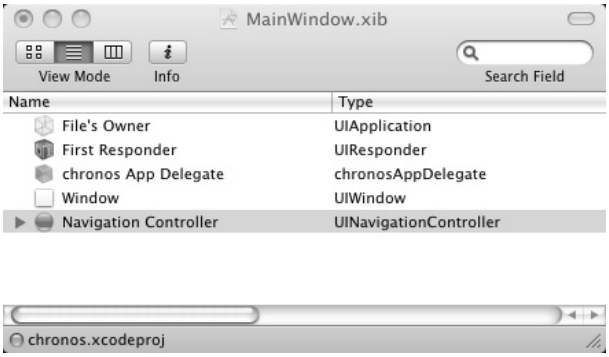


Bild 11.3: Neuer Navigation Controller im Document Window

In den Properties des frisch erzeugten Navigation Controllers (den Property-Inspektor gegebenenfalls mit *Tools > Inspector* öffnen) wird im Reiter *Attributes* der Eintrag *Shows Toolbar* ausgewählt. Die Anwendung erhält dadurch die im Kapitel »User Interface Design« diskutierte Tool Bar am unteren Displayrand, über die später verschiedene Aktionen ausgelöst werden können.

Der Navigation Controller existiert nun als Instanz im Nib-File. Um ihn in der App verwenden zu können, muss er zunächst über Instanzvariable und Outlet im Code verfügbar gemacht und anschließend dem Haupt-Window des Programms zugefügt werden.

Die Deklaration von Instanzvariable und Outlet erfolgt in der Header-Datei von `chronosAppDelegate` an bekannter Stelle.

```
...
   UIWindow *window;
   IBOutlet UINavigationController *navigationController;
}
...
```

Nun muss die neue Instanzvariable `navigationController` über das Outlet mit der in Interface Builder erzeugten Instanz des Navigation Controllers verbunden werden. Dies geschieht wiederum im Interface Builder im Document Window von `MainWindow.xib`.

Ein Rechtsklick auf *chronosAppDelegate* öffnet ein Popup-Fenster mit den verfügbaren Outlets der Klasse. Durch Klicken auf den kleinen Kreis neben *navigationController* und anschließendes Ziehen auf den Navigation Controller im darunterliegenden Fenster wird die Verbindung hergestellt:

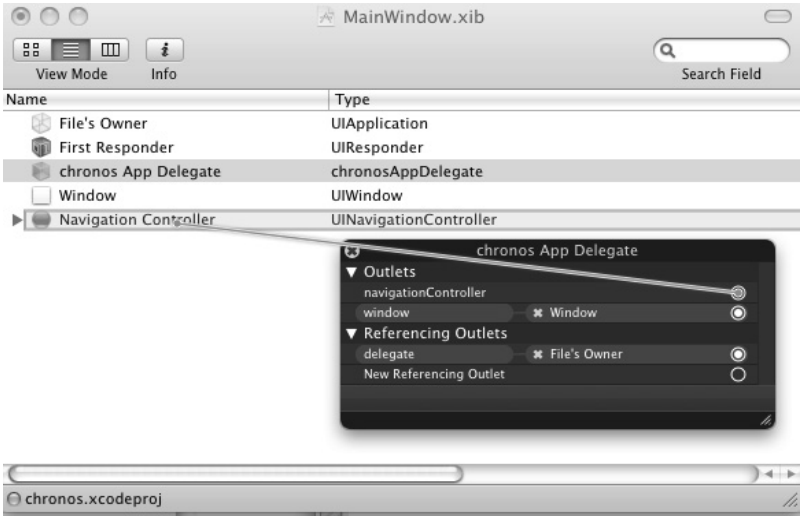


Bild 11.4: Navigation Controller Connection

Das Erzeugen der Connection kann natürlich alternativ auch im gleichnamigen Inspektor geschehen.

Durch das Anlegen der Verbindung liegt nun im Code eine gültige Referenz auf die Instanz des Navigation Controllers im Nib-File vor. Man beachte, dass – wie bei den Container-View Controllern üblich – keine Subklasse von `UINavigationController` benötigt wird.

Zu guter Letzt wird in der Klasse `chronosAppDelegate` der aktuelle View des Navigation Controllers dem Programmfenster als Subview zugefügt:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
   [window addSubview:[navigationController view]];
   [window makeKeyAndVisible];
}
```

Ein Start der App im Simulator zeigt das Resultat dieser Schritte:

Bild 11.5: Navigation Controller in der App

Die Navigation Bar und die Tool Bar sind wie gewünscht zu sehen, der Content-Bereich jedoch ist noch leer. Das ist bei genauerer Betrachtung der Datenstruktur unterhalb des Navigation Controllers allerdings nicht weiter verwunderlich:

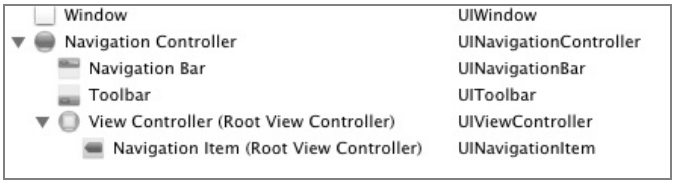


Bild 11.6: Navigation Controller mit Default-View Controller

Wie man in der Abbildung erkennen kann, enthält der Navigation Controller neben der Navigation Bar und der Tool Bar einen *Default-View Controller*. Dieser wird beim Drag & Drop aus der Library mit erzeugt. Eine Kontrolle der Outlets dieses Controllers im Connection Inspector zeigt, dass das `view`-Outlet mit keinem View verbunden ist. Folglich ist auch nichts zu sehen.

Zur Übung können Sie zum Beispiel einen *Text View* aus der Library in die Hierarchie unterhalb des *Root View Controllers* (also auf Höhe des *Navigation Item*) einfügen. Ein erneuter Start zeigt nun den Text View mit seinem Beispieltext zwischen Navigation Bar

und Tool Bar an. Vor dem Fortfahren sollte der Text View allerdings wieder gelöscht werden.

Richtig spannend sind Navigation Controller erst in Kombination mit *Table View Controllern*, die im nächsten Kapitel besprochen werden.

11.3 Anpassung der Navigation Bar

Ein Detail der letzten Abbildung wurde bisher noch nicht besprochen. Jeder vom Navigation Controller verwaltete View Controller verfügt über ein *Navigation Item*. Das Navigation Item beschreibt das Aussehen der Navigation Bar, wenn der zugehörige View Controller (bzw. sein View) gerade angezeigt wird. Wie die folgende Abbildung verdeutlicht, sind über das Navigation Item die verschiedenen Bestandteile der Navigation Bar, wie *Bar Button Items* und *Title View*, erreichbar:

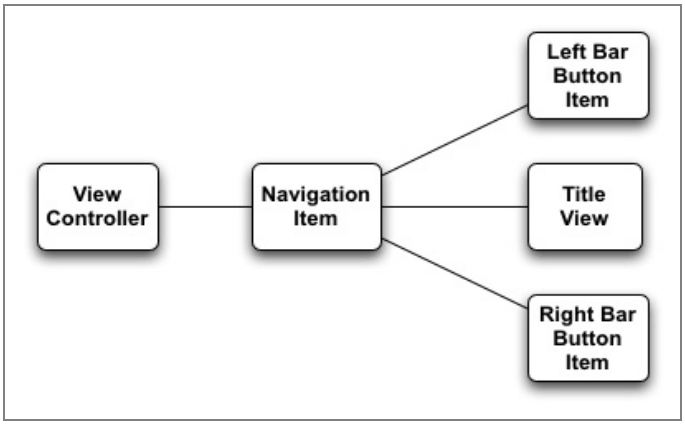


Bild 11.7: Navigation Item

11.3.1 Bar Button Items

Die Navigation Bar bietet links und rechts neben dem Titel Platz für zwei Buttons. Analog zur Verwendung von Buttons auf der Toolbar sollten bei Betätigen der Schaltflächen Aktionen aufgerufen werden, die sich auf den ganzen View und nicht auf Teilbereiche, wie etwa eine Tabellenzeile, beziehen.

Das Erstellen von Buttons mit Text oder Grafiken ist mit den beiden Methoden `initWithImage:style:target:action:` oder `initWithTitle:style:target:action:` einfach zu bewerkstelligen. Neben dem Bild oder dem Text werden einer der vordefinierten Stile `UIBarButtonItemStylePlain`, `UIBarButtonItemStyleBordered` oder `UIBarButtonItemStyleDone` sowie die aufzurufende Methode definiert. Die Methode wird über ein Target-Objekt (im Beispiel die eigene Klasse) sowie über eine Aktion identifiziert.

```
UIBarButtonItem *infoItem = [[[UIBarButtonItem alloc]
   initWithImage: [UIImage imageNamed:@"info_small.png"]
   style:UIBarButtonItemStylePlain
   target:self action:@selector(showInfo)]autorelease];
```

Nach dem Erzeugen des *Bar Button Item* wird es in einem View Controller als rechtes Bar Button Item des zugehörigen Navigation Item gesetzt und kann anschließend freigegeben werden:

```
self.navigationItem.rightBarButtonItem = infoItem;
[infoItem release];
```

Neben diesen selbst definierten Buttons gibt es eine Reihe von Bar Button System Items, die über die Methode `initWithBarButtonSystemItem:target:action:` erstellt werden können. Die System Items verfügen bereits über Bilder oder Texte. Die folgenden Items sind verwendbar:

- `UIBarButtonSystemItemDone`
- `UIBarButtonSystemItemCancel`
- `UIBarButtonSystemItemEdit`
- `UIBarButtonSystemItemSave`
- `UIBarButtonSystemItemAdd`
- `UIBarButtonSystemItemFlexibleSpace`
- `UIBarButtonSystemItemFixedSpace`
- `UIBarButtonSystemItemCompose`
- `UIBarButtonSystemItemReply`
- `UIBarButtonSystemItemAction`
- `UIBarButtonSystemItemOrganize`
- `UIBarButtonSystemItemBookmarks`
- `UIBarButtonSystemItemSearch`
- `UIBarButtonSystemItemRefresh`
- `UIBarButtonSystemItemStop`
- `UIBarButtonSystemItemCamera`
- `UIBarButtonSystemItemTrash`
- `UIBarButtonSystemItemPlay`
- `UIBarButtonSystemItemPause`
- `UIBarButtonSystemItemRewind`
- `UIBarButtonSystemItemFastForward`
- `UIBarButtonSystemItemUndo`
- `UIBarButtonSystemItemRedo`

11.3.2 Title View

Der Titel des aktuellen View Controllers, der in der Mitte der Navigation Bar angezeigt wird, kann durch jeden anderen View ersetzt werden. Ein typisches Beispiel dafür ist die Anzeige eines *Segmented Control* in der Navigation Bar, etwa zum Sortieren von Tabellendaten:

Bild 11.8: Segmented Control als Title View

Im zugehörigen Code werden nach dem Erzeugen des Segmented Control der Stil und das ausgewählte Segment gesetzt. Anschließend wird die Methode *toggleSorting:* definiert, die beim Klick auf ein Segment aufgerufen werden soll. Schließlich wird das Segmented Control als *titleView* des Navigation Item gesetzt:

```
UISegmentedControl *sortControl = [[UISegmentedControl alloc] initWithItems:
   [NSArray arrayWithObjects:@"Datum", @"Projekt", @"Leistung", nil]];
sortControl.segmentedControlStyle = UISegmentedControlStyleBar;
sortControl.selectedSegmentIndex = 0;
[sortControl addTarget:self action:@selector(toggleSorting:)
   forControlEvents:UIControlEventValueChanged];
self.navigationItem.titleView = sortControl;
```

11.3.3 Back Button

Der *Back*-Button auf der linken Seite der Navigationsleiste trägt bereits automatisch den Titel des View Controllers, zu dem der Klick auf den Button führt. Dennoch kann es gute Gründe geben, den Text ändern zu wollen. Ein Beispiel dafür ist ein langer Titel, der den Button zu groß werden lässt. Der Text kann durch Setzen eines neuen Bar Button Item einfach geändert werden, wie folgendes Codefragment zeigt:

```
UIBarButtonItem *menuBarButtonItem = [[UIBarButtonItem alloc] init];
menuBarButtonItem.title = @"Menü";
self.navigationItem.backBarButtonItem = menuBarButtonItem;
[menuBarButtonItem release];
```

11.4 Weitere Methoden zur Stapelverwaltung

Die Methoden `pushViewController:animated:` zum Auflegen eines View Controllers auf den Stapel sowie `popViewController:animated:` zum Abnehmen des obersten View Controllers wurden bereits besprochen. Darüber hinaus existieren einige weitere ganz nützliche Methoden, die kurz vorgestellt werden sollen.

Mit der Methode `popToRootViewControllerAnimated:` können alle View Controller bis auf den untersten (also zuerst zugefügten) Controller vom Stapel geworfen werden. Der unterste View Controller im Stapel wird auch als *Root View Controller* bezeichnet.

`popToViewController:animated:` erlaubt das Abnehmen der Controller im Stapel bis zu einem bestimmten View Controller, der als Parameter übergeben werden muss.

Die beiden genannten Methoden ermöglichen also ein Springen in der Navigation. Ein Wechsel auf das Hauptmenü nach einem Klick auf einen entsprechenden Button könnte damit technisch leicht realisiert werden. Es sollte mit dieser Möglichkeit allerdings sehr vorsichtig umgegangen werden, weil der Benutzer durch allzu gewagte View-Wechsel leicht verwirrt werden kann. Denken Sie an das mentale Navigationsmodell des Anwenders, das dieser sich unbewusst von der App macht ...

12 Table View Controller

Table Views werden zum Anzeigen von vertikal scrollbaren Tabellen benutzt. Da die Tabellen nur über eine Spalte verfügen, sollte man eigentlich eher von Listen sprechen. Die Zeilen bzw. Zellen der Liste können durch Sektionen gruppiert werden.

Die Anwendungsmöglichkeiten für Table Views sind vielfältig, eigentlich kommt kaum eine App ohne sie aus. Im Zusammenspiel mit einem Navigation Controller erlauben Tabellen die Darstellung komplexer hierarchischer Zusammenhänge. Ein Beispiel hierfür ist die Mail-Anwendung von Apple.

Im Projekt *Chronos* werden Table Views für das Menü und die anschließende Darstellung der Listen von Kunden, Projekten usw. verwendet. Schließlich wird sogar der Detail-View mithilfe der Tabellenkomponente realisiert.

12.1 Aufbau von Table Views

Den grundlegenden Aufbau von Table Views zeigt die folgende Abbildung:

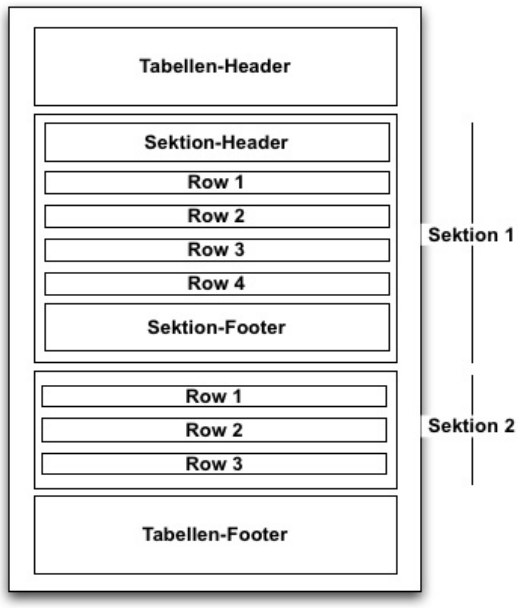


Bild 12.1: Aufbau von Table Views

Ein Table View kann einen Kopf- (*Header)* und einen Fußbereich (*Footer)* enthalten. Beim Scrollen werden diese Elemente mit fortbewegt. Der Bereich zwischen Header und Footer kann (muss aber nicht) aus mehreren Sektionen bestehen. Sektionen erlauben das Gruppieren der enthaltenen Zeilen (*Rows*). Jede Sektion kann wiederum einen Header und Footer enthalten. Verzichtet man auf alle Header und Footer und beschränkt sich auf eine einzige Sektion, ergibt sich daraus eine einfache Liste.

Table Views können in zwei Stilvarianten dargestellt werden: *Plain* oder *Grouped*. Der Grouped-Stil wird verwendet, um zusammengehörige Informationen in einer Gruppe zu präsentieren. Optisch wird die Gruppe durch abgerundete Ecken abgegrenzt. Ein ansprechendes Beispiel hierfür liefert die App *DB Navigator* der Deutschen Bahn:

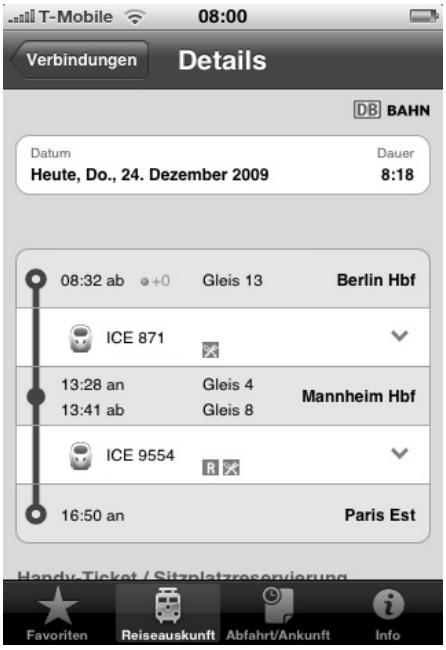


Bild 12.2: Grouped-Style Table View

Der Inhalt der einzelnen Zellen kann entweder durch Wahl eines vordefinierten Stils oder durch Implementierung einer eigenen `UITableViewCell`-Subklasse gestaltet werden.

Zur Anordnung von Texten in einer Zelle stehen vier vordefinierte Varianten zur Verfügung:

- `UITableViewCellStyleDefault`
- `UITableViewCellStyleSubtitle`
- `UITableViewCellStyleValue1`
- `UITableViewCellStyleValue2`

Die folgende Abbildung zeigt die Anordnung und das Aussehen der Labels in den verschiedenen Stilen:

Bild 12.3: Table View Cell Styles

Bei den Stilen `UITableViewCellStyleDefault` und `UITableViewCellStyleSubtitle` kann zusätzlich zu den Texten noch ein Icon auf der linken Seite der Zelle angezeigt werden.

Wie ebenfalls in der Abbildung zu erkennen ist, können alle Stile auf der rechten Zellenseite einen *Accessory View* anzeigen. Dabei handelt es sich um ein kleines Symbol, das einen Hinweis auf das Verhalten der Zelle bei einem Klick darauf gibt. Drei Standardtypen stehen zur Auswahl:

`UITableViewCellAccessoryDisclosureIndicator`	Wird verwendet, wenn die Zelle einen Drill-Down in einer Table-View-Hierarchie erlaubt.
`UITableViewCellAccessoryDetailDisclosureButton`	Wird verwendet, wenn die Auswahl der Zelle zu weiteren Details über das Zellenobjekt führt.
`UITableViewCellAccessoryCheckmark`	Wird verwendet, um zu signalisieren, dass die Zelle nach der Auswahl markiert ist.

Für viele Anwendungszwecke reichen die Gestaltungsmöglichkeiten aus. Wie Sie später sehen werden, können bei Bedarf aber auch komplett eigene Zelltypen relativ einfach definiert werden.

12.2 Erstellen von Controller & View

Der einfachste Weg zum Erzeugen einer Tabelle ist die Verwendung des Interface Builder. Die grundsätzliche Vorgehensweise sieht folgendermaßen aus:

- Erzeugen eines neuen leeren Nib-Files mit dem *New File*-Assistenten
- Erzeugen einer `UITableViewController`-Subklasse mit dem *New File*-Assistenten

- Setzen der *File's Owner* Identity Class auf die neue Subklasse
- Table View aus der Library zufügen
- `View`-Outlet des *File's Owner* per Drag & Drop auf den View ziehen

Wir wollen an dieser Stelle die Zeiterfassung ein wenig weiter vorantreiben und erstellen zunächst eine Controller-Klase für das Hauptmenü. Der Name der Klasse ist `MenuTableViewController`. Um ihrer Aufgabe als Table View Controller nachkommen zu können, muss `MenuTableViewController` eine Subklasse von `UITableViewController` sein. Dazu ist die entsprechende Option im Assistenten zu aktivieren. Eine neue Xib-Datei wird nicht benötigt, das Hauptmenü findet noch Unterschlupf in `MainWindow.xib`.:

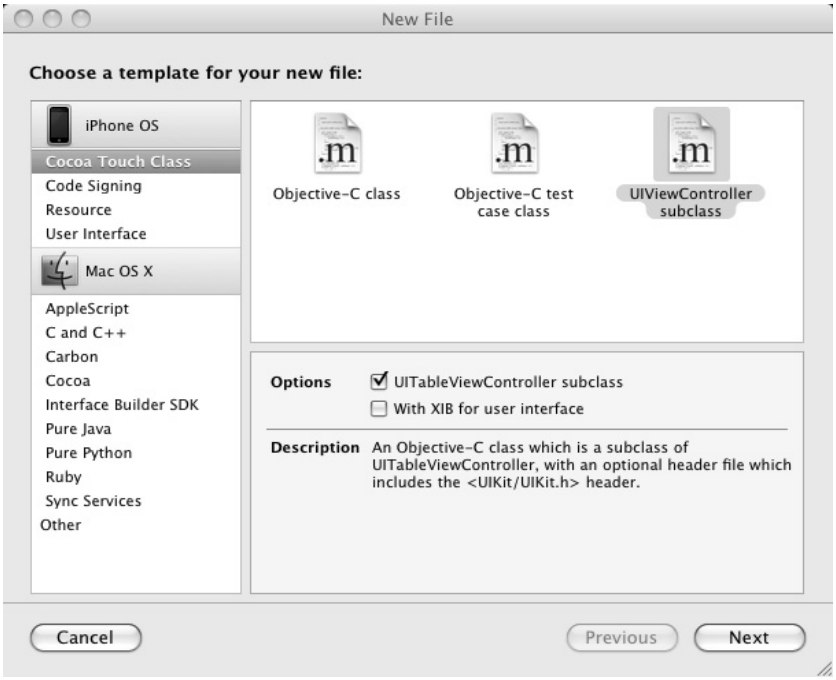


Bild 12.4: Erstellen eines Table View Controllers

Als Dateiname wird im zweiten Schritt des Assistenten `MenuTableViewController` vergeben. Alle weiteren Einstellungen sollten unverändert bleiben.

Damit die neue Klasse auch verwendet wird, wird nun im Interface Builder die Class-Identity des im Navigation Controller enthaltenen Root View Controllers auf den Namen der neuen Klasse `MenuTableViewController` gesetzt. Dadurch wird eine Instanz der neuen Klasse und nicht mehr von `UIViewController` erzeugt. Gegebenfalls muss dazu nach Markieren des Root View Controllers (dieser befindet sich in der Hierarchie unter dem Navigation Controller) der Identity-Inspektor geöffnet werden.

Ein Start der App zeigt, dass zwar noch kein Hauptmenü, aber immerhin schon eine leere Tabelle vorhanden ist:

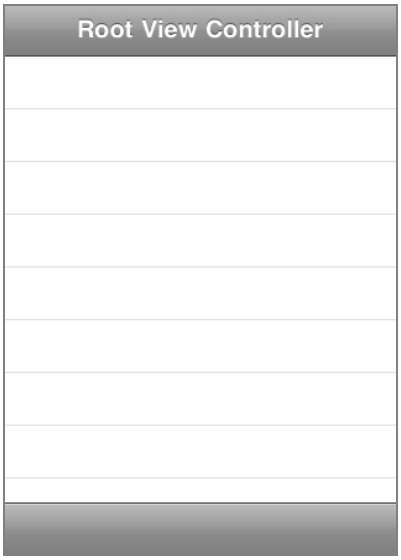


Bild 12.5: Eine erste Tabelle

Das Einfügen der einzelnen Menüpunkte ist Inhalt des nächsten Abschnitts.

12.3 Anzeigen von Daten: Data Source

Eine Tabelle ist eine Instanz von `UITableView` (oder einer ihrer Subklassen). `UITableView` verfügt über die Property `dataSource`. Über diese Variable wird der View mit dem Datenlieferanten verbunden. Die Klasse, die die Daten liefert, muss das Protokoll `UITableViewDataSource` implementieren. Damit ist sie verpflichtet, zumindest die Methoden `tableView:numberOfRowsInSection:` und `tableView:cellForRowAtIndexPath:` zur Verfügung zu stellen. Daneben enthält das Protokoll eine Reihe von optionalen Methoden. Die bevorzugte Klasse zur Implementierung des Protokolls `UITableViewDataSource` ist natürlich der *Table View Controller*. Die nächste Abbildung veranschaulicht die Zusammenhänge:

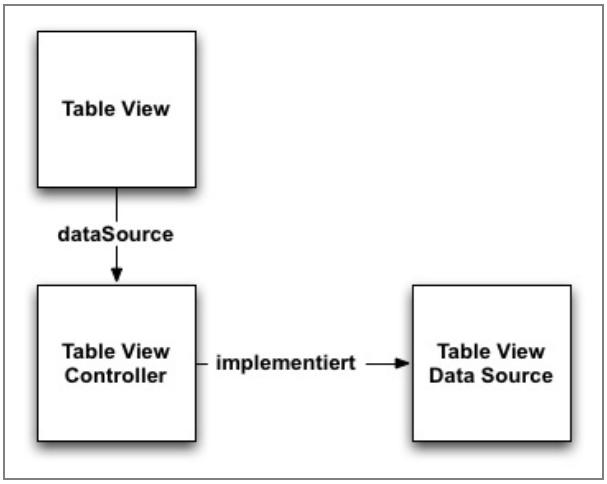


Bild 12.6: Table View, Controller und Data Source

Die wichtigsten Methoden des Protokolls werden nun kurz vorgestellt. Danach wird mit dem frisch erworbenen Wissen das Menü fertiggestellt.

12.3.1 numberOfSectionsInTableView:

Mithilfe dieser Methode wird die Anzahl der Sektionen einer Tabelle definiert. Die Methode ist optional. Wird sie nicht implementiert, besteht die Tabelle automatisch aus einer einzigen Sektion.

12.3.2 tableView:numberOfRowsInSection:

Die Methode `tableView:numberOfRowsInSection:` dient der Angabe der Zeilenanzahl der Tabelle. Da die Tabelle aus mehreren Sektionen bestehen kann, wird die Nummer der Sektion als Methodenparameter übergeben. Abhängig von der Sektion können so unterschiedlich viele Zeilen zurückgegeben werden. Das folgende Codestück demonstriert das Prinzip:

```
- (NSInteger)tableView:(UITableView *)tableView
   numberOfRowsInSection:(NSInteger)section {
   if (section==0)
     return 1;    // Sektion 0 hat eine Zeile
   else
     return 4;    // andere Sektionen haben vier Zeilen
}
```

12.3.3 tableView:cellForRowAtIndexPath:

Für jede Zeile, die angezeigt werden soll, wird die Methode `tableView:cellForRowAtIndexPath:` aufgerufen. In ihr wird die Zelle definiert und zurückgegeben.

Die mit dem Assistenten erzeugte Default-Implementierung der Methode sieht folgendermaßen aus:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
   cellForRowAtIndexPath:(NSIndexPath *)indexPath {

   static NSString *CellIdentifier = @"Cell";

   UITableViewCell *cell = [tableView
     dequeueReusableCellWithIdentifier:CellIdentifier];
   if (cell == nil) {
     cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
       reuseIdentifier:CellIdentifier] autorelease];
   }

   // Set up the cell...

   return cell;
}
```

Obwohl das die denkbar einfachste Implementierung ist, wirkt der Code schon ein wenig aufgebläht. Der Grund dafür ist die Wiederverwendung der Zellen. Um Zeit und Speicherplatz zu sparen, werden nur so viele Zellen erzeugt, wie auf dem Screen ohne Scrollen zu sehen sind. Verlässt eine Zelle durch Scrollen den sichtbaren Bereich, wird sie nicht verworfen, sondern in einer Queue (einer Warteschlange) gespeichert. Für die Zelle, die auf der anderen Tabellenseite durch die Scrollbewegung in den sichtbaren Bereich gerät, braucht dann kein neues Objekt erzeugt zu werden. Es wird stattdessen das Objekt aus der Queue verwendet.

Da innerhalb einer Tabelle unterschiedlich aussehende Typen von Zellen verwendet werden können, werden sie über einen *CellIdentifier* identifiziert. In obigem Codebeispiel heißt er schlicht `Cell`. Die Methode `tableView:dequeueReusableCellWithIdentifier:` versucht eine Zelle dieses Typs aus dem Speicher zu holen. Gelingt das nicht, ist der Rückgabewert der Methode `nil` und die Zelle muss wirklich neu erzeugt werden.

Auch Zellen gleichen Typs enthalten womöglich andere Grafiken oder andere Texte für enthaltene Labels. Die Zellen müssen deshalb vor der Rückgabe noch konfiguriert werden.

So weit die Theorie. Wir werden nun das Menü erstellen und dafür die neue Klasse `MenuTableViewController` den Erfordernissen anpassen. Zum Glück hat der Assistent zum Anlegen neuer Klassen mithilfe der Superklassenangabe schon die meisten Methoden überschrieben, sodass diese nur noch mit dem passenden Inhalt gefüllt werden müssen.

An dieser Stelle ein kleiner Tipp: Beim Überschreiben bzw. Implementieren von Methoden spart man viel Zeit und Nerven, wenn man die langen, komplizierten Methodendeklarationen einfach aus der Dokumentation heraus in den Code kopiert. Für

die Methode `tableView:numberOfRowsInSection:` beispielsweise einfach die Dokumentation öffnen, den Begriff `UITableViewDataSource` in das Suchfenster eingeben und aus der gefundenen Dokumentation die Definition der Methode (in der Abbildung grau hinterlegt) herauskopieren:

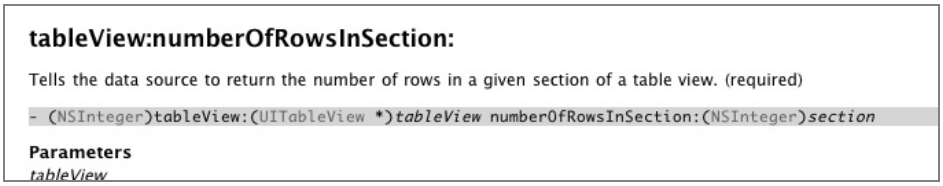


Bild 12.7: Den Methodennamen aus der Dokumentation herauskopieren

In unserem Fall ist die Methode aber ohnehin bereits vom Assistenten angelegt worden. Das zu erstellende Menü besteht aus vier Menüpunkten. Diese entsprechen den Zeilen einer einzigen Sektion. Die Anzahl der Zeilen pro Sektion wird, wie besprochen, in der Methode `tableView:numberOfRowsInSection:` definiert:

```
- (NSInteger)tableView:(UITableView *)tableView
   numberOfRowsInSection:(NSInteger)section{
   return 4;
}
```

Anschließend muss dafür gesorgt werden, dass in den vier Zeilen auch die passenden Texte für die Menüpunkte erscheinen. Wie diskutiert, wird der Inhalt einer Tabellenzeile in der Methode `tableView:cellForRowAtIndexPath:` beschrieben. Abhängig von der Zeilennummer (`indexPath.row`) wird in der Methode durch ein `switch`-Statement das Label der Zelle gesetzt:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]
autorelease];
    }

   // Set up the cell...
   cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
   switch (indexPath.row) {
     case 0: {
       cell.textLabel.text = @"Kunden";
     } break;
```

```
    case 1: {
      cell.textLabel.text = @"Projekte";
    } break;
    case 2: {
      cell.textLabel.text = @"Leistungen";
    } break;
    case 3: {
      cell.textLabel.text = @"Zeiten";
    } break;
  }

   return cell;
}
```

So wie mit `indexPath.row` auf die aktuelle Zeile zugegriffen werden kann, so gestattet übrigens der Ausdruck `indexPath.section` die Ermittlung der aktuellen Sektion.

Das Setzen des `accessoryType` der Tabellenzelle auf `UITableViewCellAccessory DisclosureIndicator` im Code oben führt zur Anzeige des Pfeils rechts neben dem Text, der auf das Drill-Down-Verhalten beim Anklicken der Zelle hinweist.

Nach dem Starten der Anwendung sieht der View nun folgendermaßen aus:

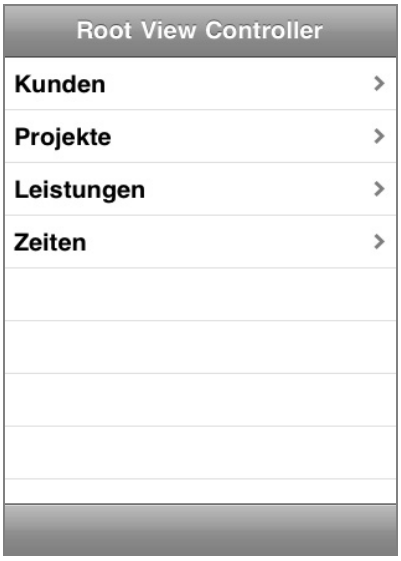


Bild 12.8: Eine erste Version des Hauptmenüs

Das sieht schon ganz brauchbar aus. Im Moment reagieren die Zellen allerdings bei Berührung nur mit einer Blaufärbung und nicht mit dem gewünschten Drill-Down-Effekt. Zugegebenermaßen wurde auch noch keine Arbeit in dieses Feature investiert.

Bevor wir uns mit Reaktionen auf Zellenklicks und dem dazugehörigen Protokoll `UITableViewDelegate` beschäftigen, ist es sinnvoll, zunächst die Tabellen zu erstellen, zu denen der Drill-Down erfolgen soll. Dabei bietet sich die Gelegenheit, die besprochenen Methoden aus `UITableViewDataSource` mit etwas komplexerem Inhalt zu füllen.

Wie man den Skizzen aus dem Kapitel »User Interface Design« entnehmen kann, verbergen sich hinter den Menüpunkten Views mit Listen von Kunden, Projekten, Leistungen und Zeiten. Die Daten, die dort angezeigt werden sollen, sind also nicht wie beim Menü statische Daten, sondern vom Benutzer angelegte Daten aus der Datenbank.

Zunächst wird der *Table View Controller* für die Kunden-Liste erstellt. Dazu wird in der bereits beschriebenen Weise ein neuer Table View Controller, also eine Subklasse von `UITableViewController`, zugefügt. Der Name der Klasse ist `KundenTableViewController`. Allerdings sollte diesmal auch die Option *With XIB for user interface* markiert sein. Das durch diese Aktion erzeugte Xib-File `KundenTableViewController.xib` wird nach dem Erzeugen in `KundenListeView.xib` umbenannt und aus Gründen der Übersichtlichkeit in die Gruppe *Resources* im Xcode-Projekt verschoben.

Für die weiteren Listen-Controller wird analog verfahren; die Namen sind `ProjekteTableViewController`, `LeistungenTableViewController` und `ZeitenTable ViewController`.

Wie kommen nun die Daten aus der Datenbank in die Tabelle? Offensichtlich müssen wieder die Methoden `tableView:numberOfRowsInSection:` und `tableView: cellForRowAtIndexPath:` in der neuen Klasse implementiert werden. In ersterer muss die Anzahl der Kunden-Datensätze in der Datenbank ermittelt und zurückgegeben werden. In der zweiten Methode müssen die Namen der Kunden in die Labels der Zellen gesetzt werden.

Um komfortabel auf die DB-Daten zugreifen zu können, wird ein Objekt vom Typ `NSFetchedResultsController` benutzt. Spezialgebiet dieser Klasse ist das Zur-Verfügung-Stellen von *Core Data*-Daten in Table Views. Die Klasse kann zum Beispiel dabei helfen, die geladenen Daten in Sektionen zu unterteilen. Das Erstellen des Objekts ist relativ aufwendig. Damit es nur einmal erzeugt werden muss, wird es in einer Instanzvariablen vorgehalten. Zunächst werden also Instanzvariable und Property für ein Objekt `fetchedResultsController` vom Typ `NSFetchedResultsController` im neuen `KundenTableViewController` angelegt:

```
@interface KundenTableViewController : UITableViewController {

   NSFetchedResultsController *fetchedResultsController;
}

@property (nonatomic, retain) NSFetchedResultsController
*fetchedResultsController;
@end
```

Die Variable wird außerdem im Implementierungs-File »synthetisiert« und in `dealloc` mit `release` wieder freigegeben. Wie Sie sich vielleicht erinnern, werden durch

`@synthesize` die `get`- und `set`-Methoden erzeugt, sofern keine Implementierung von Hand vorhanden ist. Genau diese wird nun aber für den Accessor, also die `get`-Methode erstellt:

```
- (NSFetchedResultsController *)fetchedResultsController {

   chronosAppDelegate*appDelegate = (chronosAppDelegate*)[[UIApplication
     sharedApplication] delegate];

   if (fetchedResultsController != nil) {
     return fetchedResultsController;
   }

   NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
   NSEntityDescription *entity = [NSEntityDescription entityForName:@"Kunde"
     inManagedObjectContext:appDelegate.managedObjectContext];
   [fetchRequest setEntity:entity];

   NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
     initWithKey:@"name" ascending:YES];
   NSArray *sortDescriptors = [[NSArray alloc]
initWithObjects:sortDescriptor, nil];
   [fetchRequest setSortDescriptors:sortDescriptors];

   NSFetchedResultsController *aFetchedResultsController =
     [[NSFetchedResultsController alloc] initWithFetchRequest:fetchRequest
     managedObjectContext:appDelegate.managedObjectContext
     sectionNameKeyPath:nil cacheName:@"Root"];
     self.fetchedResultsController = aFetchedResultsController;

   [aFetchedResultsController release];
   [fetchRequest release];
   [sortDescriptor release];
   [sortDescriptors release];

   return fetchedResultsController;
}
```

Zunächst wird über die bereits vorgestellte Klassenmethode `sharedApplication` der Klasse `UIApplication`, gefolgt vom Aufruf `delegate`, die Instanz der `Application Delegate`-Klasse ermittelt und in eine Variable gepackt.

Danach folgt eine Überprüfung, ob der Instanzvariablen `fetchedResultsController` eventuell schon ein Objekt zugewiesen wurde. Das ist ab dem zweiten Aufruf der Methode der Fall. Die Methode wird dann an dieser Stelle beendet.

Beim ersten Aufruf der Methode wird als Nächstes ein *Fetch Request* erzeugt. Die Klasse wurde im Kapitel »Core Data« bereits besprochen. Damit dem Fetch Request bekannt ist, welche Daten er aus der Datenbank holen soll, wird er mit einer Art Steckbrief, einer

Entity Description, ausgestattet. Im Quellcode oben soll er nach Kunden suchen. Man beachte, dass zum Ermitteln der Description auf den ebenfalls bereits diskutierten *Managed Object Context* im Application Delegate zugegriffen wird. Dort sind die Beschreibungen für alle Entitäten hinterlegt.

Als Nächstes folgt im Code die im Moment nicht ganz so wichtige Definition der Sortierreihenfolge für die zu lesenden Daten.

Schließlich wird mithilfe des Fetch Request ein Objekt vom Typ `NSFetchedResultsController` erzeugt, der Instanzvariablen zugewiesen und als Rückgabewert der Methode zurückgegeben.

Weiterhin fehlt noch der Import von `chronosAppDelegate`:

```
#import "chronosAppDelegate.h"
```

Damit der `fetchedResultsController` auch wirklich Daten von der Datenbank holt, bekommt er die Nachricht `performFetch` gesendet. Ein guter Platz dafür ist die `viewDidLoad-Methode` im `KundenTableViewController`:

```
- (void)viewDidLoad {
   [super viewDidLoad];
   NSError *error;
   if (![[self fetchedResultsController] performFetch:&error]) {
     // Handle the error...
   }
}
```

Der Zugriff auf die Daten in der Datenbank ist damit komplett und die Methoden aus dem `UITableViewDataSource`-Protokoll können implementiert werden.

In `tableView:numberOfRowsInSection:` wird statt einer hart codierten Zahl einfach der Aufruf `[[fetchedResultsController fetchedObjects] count]` zurückgegeben:

```
- (NSInteger)tableView:(UITableView *)tableView
   numberOfRowsInSection:(NSInteger)section {
   return [[fetchedResultsController fetchedObjects] count];
}
```

Die Methode `fetchedObjects` liefert alle gefundenen Objekte als `NSArray` zurück. `count` zählt die Objekte im Array. Das Resultat ist somit die Anzahl der Kunden.

Ähnlich einfach ist die Implementierung von `tableView:cellForRowAtIndexPath:`. Dort wird mithilfe der aktuellen Zeile, zugreifbar über `indexPath`, und der Methode `objectAtIndexPath:` ein Kunde als *Managed Object* ermittelt. Die *Property* `Name` dieses Objekts wird als Text für das Zellenlabel gesetzt. Ergebnis ist also eine Liste mit Kundennamen. Beachten Sie den KVC-Zugriff auf die Property:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";
```

```
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]
autorelease];
    }

    // Set up the cell...
   NSManagedObject *managedObject = [fetchedResultsController
     objectAtIndexPath:indexPath];
   cell.textLabel.text = [[managedObject valueForKey:@"name"] description];

    return cell;
}
```

Damit ist die Kundenliste vorerst abgeschlossen. Die Implementierung der anderen Listen-Klassen erfolgt analog und soll hier nicht erneut wiedergegeben werden.

Unglücklicherweise kann man das Ergebnis dieser doch komplexeren Arbeiten noch nicht begutachten. Es fehlt noch das Stück Code, das den Übergang vom Menü zur Kundenliste ermöglicht. Im nun folgenden Abschnitt wird es besprochen.

12.4 Aktionen in Tabellen: Delegate

Das zweite wichtige Protokoll im Tabellenbereich heißt `UITableViewDelegate` und ist zuständig für Aussehen und Interaktionsverhalten der Tabelle. Im Einzelnen sind Methoden für die folgenden Bereiche enthalten:

- Höhe von Zellen, Header und Footer
- Selektion von Zellen oder *Accessory Views*
- Views für Header und Footer
- Typ des Accessory View
- *Edit Mode*-Verhalten
- Ändern der Zellenreihenfolge

Die Klasse, die das Protokoll implementiert, ist, wie bei `UITableViewDataSource`, im Normalfall ebenfalls der Table View Controller. Er wird dem Table View über die Property `delegate` gesetzt.

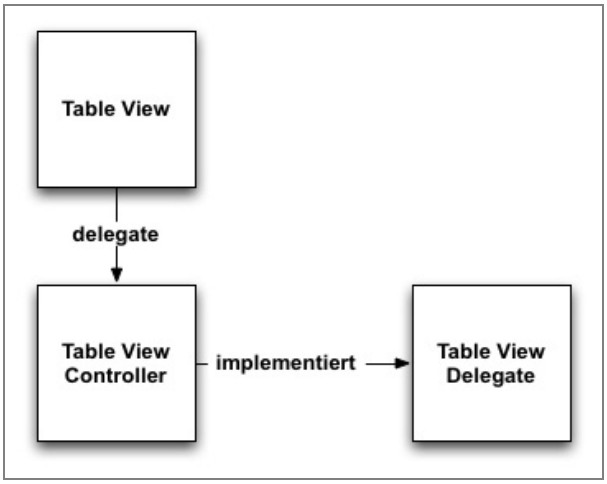


Bild 12.9:
Controller und Delegate

Die Methode `tableView:didSelectRowAtIndexPath` ist vermutlich die meistbenutzte aus dem Protokoll. Sie beschreibt das Verhalten der App, wenn der Benutzer auf eine Zelle der Tabelle klickt. Im Duo *Table View Controller/Navigation Controller* wird hier also der oft erwähnte Drill-Down ausprogrammiert.

Um einen Blick auf die Kundenliste werfen zu können, benötigen wir eine Implementierung für diese Methode in der Klasse `MenuTableViewController`. Die einfachste denkbare Codierung sieht folgendermaßen aus:

```
- (void)tableView:(UITableView *)tableView
   didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
   KundenTableViewController *kundenTableViewController =
     [[KundenTableViewController alloc]
     initWithNibName:@"KundenListeView"
     bundle:nil];
   [self.navigationController pushViewController:kundenTableViewController
     animated:YES];
   [kundenTableViewController release];
}
```

Hier wird eine Instanz des `KundenTableViewController` unter Angabe des dazugehörigen Nib-Files (`KundenListeView.xib`) erzeugt und auf den Stapel der vom Navigation Controller verwalteten View Controller gelegt. Der `KundenTableView-Controller` wird damit oberster View Controller im Stapel und der dazugehörige View sichtbar.

Der Code funktioniert zwar, ist aber nicht sonderlich performant. Das Erstellen des Table View Controllers wird bei jedem Drill-Down ausgeführt. Das ist natürlich unnötig, die Methode sollte noch ein wenig verbessert werden.

Dazu wird im Header-File der Klasse `MenuTableViewController` eine Instanzvariable vom Typ `KundenTableViewController` als Property definiert:

```
#import <UIKit/UIKit.h>

@class KundenTableViewController;

@interface MenuTableViewController : UITableViewController {
   KundenTableViewController *kundenTableViewController;
}

@property (nonatomic, retain) KundenTableViewController
*kundenTableViewController;

@end
```

Die `@class`-Anweisung im HeaderFile wurde bisher noch nicht besprochen. Sie liefert dem Compiler die Information, dass es sich bei `KundenTableViewController` um eine Klasse handelt. In den Header-Files sollte zur Vermeidung zyklischer Abhängigkeiten `@class` an Stelle von `#import` verwendet werden.

In der Implementierung von `MenuTableViewController` wird eine Methode zur Rückgabe der `KundenTableViewController`-Instanz hinzugefügt. Beim ersten Aufruf der Methode wird das Objekt mit `alloc` erzeugt und danach mit `initWithNibName` initialisiert. Dabei wird der Name der Xib-Datei übergeben. Bei weiteren Aufrufen der Methode wird das Objekt direkt zurückgegeben und das Nib-File braucht nicht erneut eingelesen zu werden:

```
- (KundenTableViewController *)kundenTableViewController {
    if (kundenTableViewController == nil) {
        kundenTableViewController = [[KundenTableViewController alloc]
     initWithNibName:@"KundenListeView" bundle:nil];
    }
    return kundenTableViewController;
}
```

In der Methode `tableView:didSelectRowAtIndexPath:`, also beim Klicken auf die Zelle, wird das `kundenTableViewController`-Objekt nun von der zuvor erzeugten Methode angefordert und wie gehabt mit `pushViewController:animated:` auf den Stapel der Navigation Controller Views geschoben:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    // Navigation logic may go here. Create and push another view
controller.
   KundenTableViewController *controller = self.kundenTableViewController;
   [self.navigationController pushViewController:controller animated:YES];
}
```

Die folgende Abbildung zeigt die neue Tabelle nach dem Drill-Down:

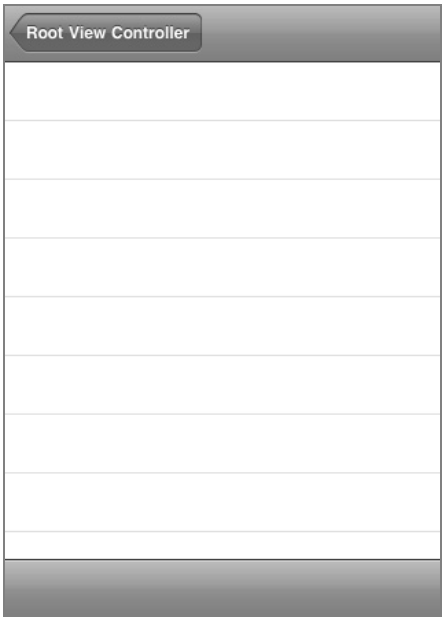


Bild 12.10: Die Kundenliste

Erstmals wird nun der Back-Button in der linken oberen Ecke sichtbar. Wie erwartet trägt er den Titel des zuvor angezeigten *View Controllers*. Der neue `KundenTableViewController` hat noch keinen Titel, die Mitte der Navigation Bar ist deshalb leer.

Außerdem werden noch keine Kunden angezeigt, die Datenbank ist noch leer. Um zu sehen, ob der Datenbankzugriff funktioniert, wären ein paar Datensätze in der Datenbank ganz praktisch. Mit der Klassenmethode `insertNewObjectForEntityForName:inManagedObjectContext:` der Klasse `NSEntityDescription` kann ein Testdatensatz für einen Entity-Typ wie einen Kunden angelegt werden. Der neue Kunde bekommt anschließend mit `setValue:forKey:` einen Namen gesetzt.

```
NSManagedObject *kunde1 = [NSEntityDescription
   insertNewObjectForEntityForName:@"Kunde"
   inManagedObjectContext:self.managedObjectContext];
[kunde1 setValue:@"Meier" forKey:@"name"];
```

Zu Testzwecken kann der Code vorübergehend beispielsweise in der Methode `applicationDidFinishLaunching:` des Application Delegates eingebaut werden. Die Kundenliste zeigt danach die ersten Kundendatensätze aus der Datenbank an:

Bild 12.11: Kundenliste mit Testdaten

Für den Menüpunkt »Kunden« funktioniert die Navigation nun. Die Vorgehensweise wird für die Klassen `ProjekteTableViewController`, `LeistungenTableViewController` und `ZeitenTableViewController` wiederholt.

Ist dies geschehen, ist eine Überarbeitung der Methode `table-View: didSelectRowAtIndexPath:` in der Klasse `MenuTableViewController` fällig. Abhängig vom ausgewählten Menüpunkt muss der passende Table View Controller für den Drill-Down verwendet werden:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
   UITableViewController *controller = nil;
   switch (indexPath.row) {
     case 0:
       controller = self.kundenTableViewController;
       break;
     case 1:
       controller = self.projekteTableViewController;
       break;
     case 2:
       controller = self.leistungenTableViewController;
       break;
     case 3:
       controller = self.zeitenTableViewController;
       break;
   }
   [self.navigationController pushViewController:controller animated:YES];
}
```

Damit ist die App schon ein ganzes Stück vorangekommen. Vom Hauptmenü aus sind die vier Listen erreichbar, die sich hinter den Menüpunkten verbergen.

12.5 Eigene Zellen

Wenn die vier vordefinierten Zelltypen den Anforderungen nicht mehr genügen, wird es Zeit, selber Hand anzulegen. Mit relativ wenig Aufwand lassen sich eigene Zellen vollkommen frei gestalten. Für unser *Chronos*-Projekt benötigt die Zeitenliste einen eigenen Zelltyp. Insgesamt sollen hier vier Texte (Datum, Dauer, Projekt und Leistung) unterge-

bracht werden. Da die vordefinierten Stile nur über maximal zwei Labels verfügen, muss eine eigene Zelle erstellt werden.

Zunächst wird im Header des View Controllers der Tabelle, für die der eigene Zelltyp entwickelt werden soll (hier `ZeitenTableViewController`), eine Instanzvariable für die Zelle als Outlet-Property definiert:

```
...
   UITableViewCell *tableViewCell;
}

@property (nonatomic, assign) IBOutlet UITableViewCell *tableViewCell;
...
```

Im Implementation-Teil werden via `synthesize` die `get`- und `set`-Methode erzeugt.

Anschließend wird aus Xcode heraus ein neues, leeres Nib-File mit dem Assistenten (*File > New File... > User Interface > Empty XIB*) angelegt. Als Name wird `ZeitenTableViewCell.xib` vergeben.

Das neue Nib-File wird per Doppelklick in Interface Builder geöffnet und aus der Library wird eine *Table View Cell* in das Document Window gezogen. Durch Doppelklick auf die gerade hinzugefügte Zelle im Document Window wird die Designoberfläche für diese geöffnet.

Wie jeder andere View kann die Zelle nun mit Labels oder anderen Komponenten bestückt werden. Die folgende Abbildung zeigt die Zelle in der Designoberfläche nach dem Zufügen der vier benötigten Labels:

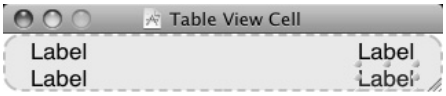


Bild 12.12: Frei gestaltete Table View Cell

Die Texte der Labels sollen später im Code mit aktuellen Daten zu der erfassten Zeit gesetzt werden. Um dort auf die einzelnen Labels zugreifen zu können, müssten nun für jedes Label Outlets im Table View Controller erzeugt werden. Wie Sie inzwischen bestimmt schon gemerkt haben, ist das recht viel Arbeit. Ein einfacherer Weg zur Identifikation der Labels im Code ist die Tag-Property. Sie findet sich im *View*-Bereich des Attribute-Inspektors bei selektiertem Label:

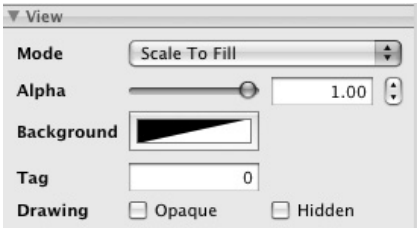


Bild 12.13: Tag-Property

Die Property wird für jedes Label auf einen anderen Zahlenwert gesetzt. Anhand der Tags können die Labels später im Code unterschieden werden.

Eine weitere wichtige Property, die bei selektierter Zelle zu definieren ist, ist der Identifier. Über ihn kann der Zelltyp später für das Caching identifiziert werden. Er wird auf *ZeitenListeCell* gesetzt:

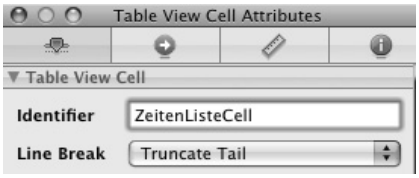


Bild 12.14: Identifier-Property

Die Identity Class des *File's Owner* im Document Window wird auf den View Controller gesetzt, der das Nib-File laden wird. Er ist der Besitzer der Objekte in der Datei. Hier ist das `ZeitenTableViewController`. Durch das Setzen des *File's Owner* wird das anfangs definierte Outlet in Interface Builder verfügbar. Ein Rechtsklick auf *File's Owner* zeigt das *Connection*-Popup an, auf dem sich die Verbindung zwischen Outlet und Komponente erstellen lässt:

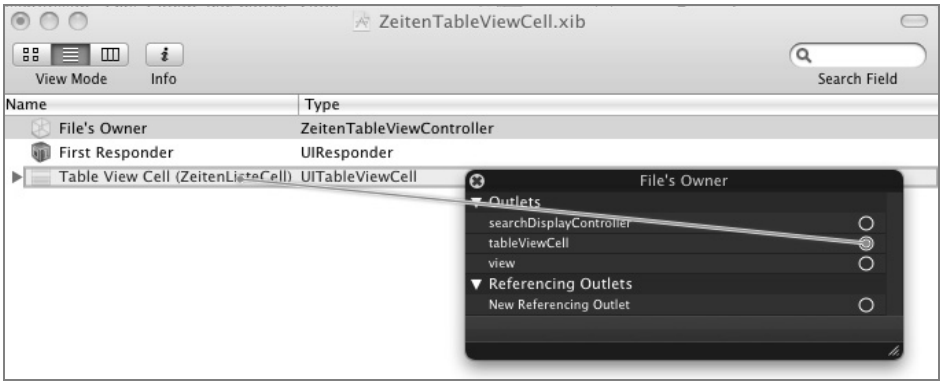


Bild 12.15: Connection für die Table View Cell

Die Arbeit im Interface Builder ist damit abgeschlossen. Das Laden der neuen Zelle aus dem Nib-File und das Setzen der Texte wird im Code in der Methode `tableView:cellForRowAtIndexPath:` implementiert. Wie im folgenden Quellcode ersichtlich ist, wird zunächst mithilfe des Identifiers versucht, eine Zelle vom Typ *ZeitenListeCell* aus der Queue zu laden. Gelingt das nicht, wird das Nib-File `ZeitenTableViewCell.xib` mit der neu definierten Zelle geladen, und die Variable `cell` bekommt die darin definierte Zelle zugewiesen:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
   cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
```

```
   static NSString *MyIdentifier = @"ZeitenListeCell";
   UITableViewCell *cell = [tableView
     dequeueReusableCellWithIdentifier:MyIdentifier];
   if (cell == nil) {
     [[NSBundle mainBundle] loadNibNamed:@"ZeitenTableViewCell" owner:self
       options:nil];
     cell = tableViewCell;
     cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
          self.tableViewCell = nil;
    }

   // Configure the cell.
   Zeit *zeit = (Zeit *)[fetchedResultsController
objectAtIndexPath:indexPath];

   UILabel *label;
   label = (UILabel *)[cell viewWithTag:1];
   label.text = [dateFormatter stringFromDate:zeit.datum];

   label = (UILabel *)[cell viewWithTag:2];
   label.text = [zeit.dauer stringValue];

   if (zeit.projekt!=nil)
   {
     label = (UILabel *)[cell viewWithTag:3];
     NSString *text = zeit.projekt.name;
     if (zeit.projekt.kunde != nil)
     {
       text = [NSString stringWithFormat:@"%@(%@)", text,
         zeit.projekt.kunde.name];
     }
     label.text = text;
   }

   label = (UILabel *)[cell viewWithTag:4];
   label.text = zeit.leistung.name;

   return cell;
}
```

Beachtenswert beim Durchsehen des Quellcodes ist außerdem die Adressierung der Labels mithilfe der Methode `viewWithTag:`. Mit ihrer Hilfe kann auf die mit Tags versehenen Labels zugegriffen werden, ohne dass dafür Outlets und Connections erstellt wurden.

12.6 Edit Mode

Alle View Controller erben die Property `editing` von der Superklasse `UIViewController`. Mithilfe dieser Variablen lässt sich der Edit Mode für einen View aktivieren oder deaktivieren. Ist der Mode aktiv, darf der View editiert werden, das heißt, es können Daten zugefügt, verändert oder gelöscht werden. Somit können mit einem einzigen View Controller sowohl die Anzeige als auch das Editieren des Inhalts realisiert werden. Die konkrete Umsetzung erfolgt in den Subklassen, etwa im Table View Controller.

Im Zusammenspiel mit der `editing`-Property unterstützt der Navigation Controller einen *Edit/Done*-Button in der Navigation Bar. Er kann mit folgender Codezeile im betroffenen View Controller (etwa in `viewDidLoad`) zur Ansicht gebracht werden:

```
self.navigationItem.rightBarButtonItem = self.editButtonItem;
```

Der Button zeigt abhängig vom Wert der Variablen die Beschriftung *Bearbeiten/Fertig* bzw. *Edit/Done* an:

Bild 12.16: Edit-Button

Ein Klick auf den Button schaltet den Modus um. Dies geschieht durch Aufruf der Methode `setEditing:animated:` des aktuellen View Controllers. Die Methode wird von `UIViewController` vererbt und kann überschrieben werden, um das Aussehen des Views für den jeweiligen Modus anzupassen.

Tabellen weisen eine sehr gute Unterstützung des Edit Mode auf. Die Default-Implementierung des Table View Controllers blendet im Edit Mode die *Löschen*-Icons (vergleichbar den roten »Einfahrt verboten«-Verkehrsschildern) auf der linken Seite der Zellen ein. Sie werden in der Zeiterfassung verwendet, um Kunden, Projekte, Leistungen oder Zeiten zu löschen:

Bild 12.17: Löschen-Icons

Neben den Icons zum Löschen von Zeilen existiert auch eine Visualisierung für das Zufügen von Zeilen. Das grüne Pluszeichen findet zum Beispiel in der *Kontakte*-App Verwendung:

Bild 12.18: Kontakte-App mit Zufügen-Icons

Sollen einzelne Zeilen von der Editierbarkeit ausgeschlossen werden, dann kann die Methode `tableView:canEditRowAtIndexPath:` überschrieben werden. Abhängig von der aktuellen Zeile (`indexPath.row`) wird ein Boole'scher Wert zurückgegeben, der angibt, ob die Zeile editierbar ist oder nicht.

Ob die *Löschen*- oder *Zufügen*-Symbole angezeigt werden, hängt dagegen von der Implementierung der Methode `tableView:editingStyleForRowAtIndexPath:` im Delegate ab. Ist die Methode nicht implementiert, werden automatisch die Icons für das Löschen von Zeilen verwendet.

Beim Klick auf das grüne Pluszeichen oder den *Delete*-Button, der dem Klick auf das rote Minuszeichen folgt, wird die Methode `tableView:commitEditingStyle:forRowAtIndexPath:` aufgerufen. In ihr wird das eigentliche Löschen oder Zufügen der Daten codiert:

```
// Override to support editing the table view.
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath
{
   if (editingStyle == UITableViewCellEditingStyleDelete) {
     NSManagedObjectContext *context = [fetchedResultsController
       managedObjectContext];
     [context deleteObject:[fetchedResultsController
       objectAtIndexPath:indexPath]];

     NSError *error;
     if (![context save:&error]) {
       // Handle the error...
     }
```

```
    [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
      withRowAnimation:UITableViewRowAnimationFade];
   }
}
```

Zu Beginn der Methode wird mithilfe des Parameters `editingStyle` zunächst geprüft, welche Aktion ausgeführt werden soll. Neben dem im Code verwendeten `UITableViewCellEditingStyleDelete` könnte auch auf `UITableViewCellEditingStyleInsert` geprüft werden. Handelt es sich um die *Delete*-Aktion, dann werden im weiteren Verlauf die betroffenen Daten sowohl aus der Datenbank als auch aus der Tabelle gelöscht. Das Entfernen der Zeilen aus der Tabelle ist Aufgabe der Methode `deleteRowsAtIndexPaths:withRowAnimation:`. Ihr muss mitgeteilt werden, mit welchem Animationseffekt die Zellen gelöscht werden sollen. Zur Auswahl stehen:

- `UITableViewRowAnimationFade`
- `UITableViewRowAnimationRight`
- `UITableViewRowAnimationLeft`
- `UITableViewRowAnimationTop`
- `UITableViewRowAnimationBottom`
- `UITableViewRowAnimationNone`

Sollen einer Tabelle Daten zugefügt werden, steht dafür das Gegenstück zu `deleteRowsAtIndexPaths:withRowAnimation:`, nämlich `insertRowsAtIndexPaths:withRowAnimation:` zur Verfügung.

12.7 Header & Footer

Wie eingangs erwähnt, können die ganze Tabelle oder auch einzelne Sektionen Header und Footer beinhalten. Der Bereich findet sich vor bzw. nach der Tabelle oder den einzelnen Sektionen und kann durch einen eigenen View visualisiert werden. Header und Footer sind ein guter Platz für diverse Buttons, eine *Search Bar* oder auch *Segmented Controls* zum Sortieren.

Im Folgenden soll die Beispiel-App einen Footer für eine erste schnelle Auswertung der erfassten Zeiten bekommen. Im Fußbereich soll die Summe der Zeiten und der Beträge dargestellt werden.

Dazu wird in Xcode mit *File > New File … > User Interface > Empty XIB* zunächst ein neues leeres Nib-File namens `FooterView.xib` angelegt. Im Interface Builder bekommt das leere Nib-File einen View aus der Library per Drag & Drop zugefügt. Der neue View ist zunächst viel zu groß. Er wird deshalb im Size-Inspektor auf eine Höhe von 100 Pixel beschränkt:

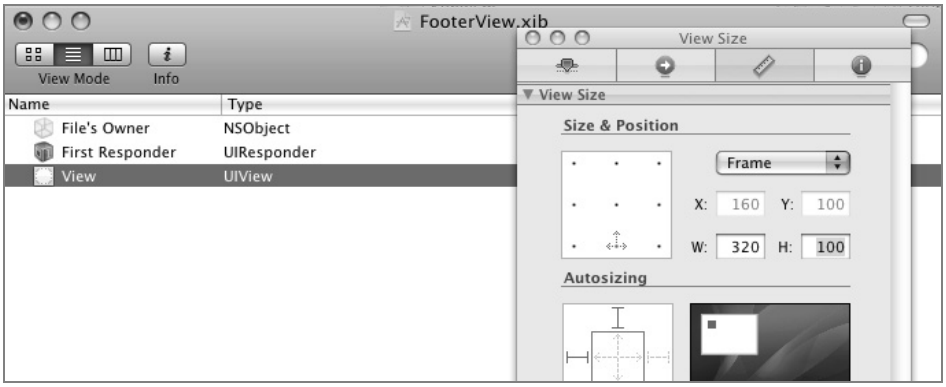


Bild 12.19: Einstellen der Footer-Höhe

Danach kann der View gestaltet werden. Für die Auswertung werden vier Labels benötigt, die aus der Library in die Designoberfläche des Views gezogen werden. Die folgende Abbildung zeigt den fertigen Footer-View:

Bild 12.20: Footer-View

Die im Moment noch mit *Label* beschrifteten Labels bekommen zur Laufzeit Texte mit den berechneten Werten für Stunden und Betrag gesetzt. Damit keine Outlets für die Labels angelegt werden müssen, werden diese wie im Abschnitt »Eigene Zellen« mit Tags gekennzeichnet. Wie schon dort besprochen, findet sich die Tag-Property bei selektiertem Label im View-Bereich des Attributes-Inspektor.

Anschließend wird der neue View mit dem Code verbunden. Die Klasse `ZeitenTableViewController` erhält dazu eine als Outlet deklarierte Instanzvariable für den Footer:

```
IBOutlet UIView *footerView;
```

Außerdem wird die Klasse im Identity Inspector des Interface Builder als Identity Class des *File's Owner* eingetragen und die Verbindung von der Variablen `footerView` des *File's Owner* zum neuen View gezogen:

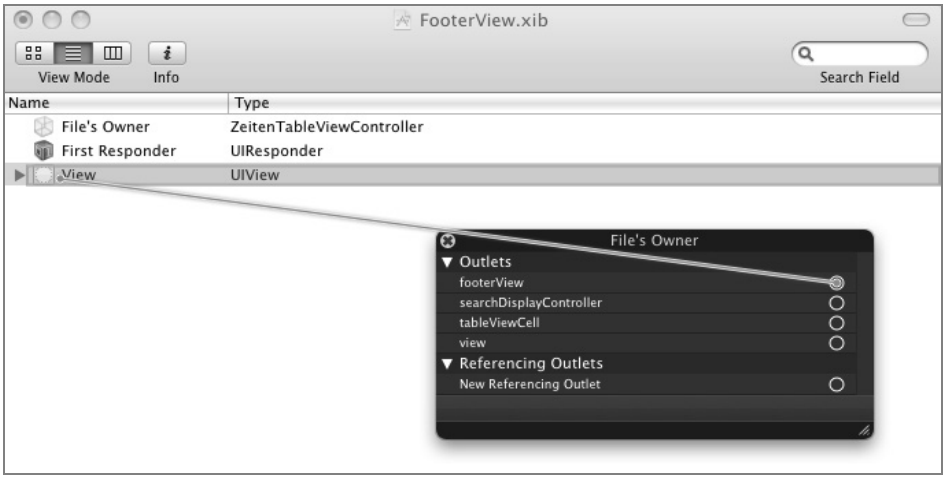


Bild 12.21: Connection für den Footer

Nun muss der neue View vom *File's Owner* geladen und als `tableFooterView` der Tabelle gesetzt werden. Dies geschieht in der Methode `viewDidLoad` von `ZeitenTableViewController`, wie der folgende Quellcode zeigt:

```
- (void)viewDidLoad
{
   [super viewDidLoad];
   self.title = @"Zeiten";

   [[NSBundle mainBundle]loadNibNamed:@"FooterView" owner:self options:nil];
   self.tableView.tableFooterView = footerView;
…
```

Der Footer der Tabelle ist damit sichtbar. Als letzte Aufgabe bleibt das Setzen von Gesamtbetrag und Gesamtdauer als Texte für die entsprechenden Labels. In der Methode `calculate` werden die Summen zunächst berechnet und anschließend den Labels zugewiesen:

```
.-(void)calculate
{
   NSDecimalNumber *gesamtDauer = [NSDecimalNumber zero];
   NSDecimalNumber *gesamtBetrag = [NSDecimalNumber zero];

   NSArray *zeiten = [fetchedResultsController fetchedObjects];
   for (Zeit *zeit in zeiten)
   {
     gesamtDauer = [gesamtDauer decimalNumberByAdding:zeit.dauer];
     if (zeit.leistung.fakturierbar.boolValue==YES &&
        zeit.leistung.stundensatz != nil)
     {
        gesamtBetrag = [gesamtBetrag decimalNumberByAdding:[zeit.dauer
```

```
        decimalNumberByMultiplyingBy:zeit.leistung.stundensatz]];
      }
    }

    UILabel *label;

    label = (UILabel *)[footerView viewWithTag:1];
    label.text = [gesamtDauer stringValue];

    label = (UILabel *)[footerView viewWithTag:2];
    label.text = [currencyFormatter stringFromNumber:gesamtBetrag];
}
```

Die Methode `calculate` wird in `viewWillAppear:` und somit bei jedem Erscheinen des Views aufgerufen.

Die folgende Abbildung zeigt den fertigen Auswertungs-Footer:

Bild 12.22: Auswertungs-Footer

Die Optik lässt sicherlich noch zu wünschen übrig, das soll hier aber nicht weiter interessieren.

12.8 Die Detail-Views

Wie zu Beginn des Kapitels erwähnt, werden auch die Detail-Views mithilfe der Tabellenkomponente realisiert. Sie erinnern sich hoffentlich noch an die Skizzen im Kapitel »User Interface Design«. Die Detail-Views zeigen die Daten nach einem Klick auf einen Listeneintrag (etwa einen Kunden) auf einem weiteren View im Detail an. Beim Kunden sind das Name und Bemerkung.

Die Realisierung wird hier lediglich kurz zusammengefasst. Es müssen keine neuen Konzepte besprochen werden. Alle Theorie, die Sie benötigen, um die Detail-Views zu verstehen, wurde bereits besprochen.

Der Drill-Down zum `KundenDetailViewController` erfolgt in der Klasse `KundenTableViewController`. Wie dem folgenden Code zu entnehmen ist, bekommt die Instanzvariable `kunde` des `detailViewController` vor der Anzeige einen Kunden zugewiesen. Der Kunde wird aufgrund der ausgewählten Zeile aus der Datenquelle geholt:

```
- (void)tableView:(UITableView *)tableView
   didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
   KundenDetailTableViewController *detailViewController =
     [[KundenDetailTableViewController alloc]
     initWithStyle:UITableViewStyleGrouped];
   Kunde *selectedKunde = (Kunde *)[[self fetchedResultsController]
     objectAtIndexPath:indexPath];
   detailViewController.kunde = selectedKunde;
   [self.navigationController pushViewController:detailViewController
     animated:YES];
   [detailViewController release];
}
```

In der detaillierten Kundenansicht werden die beiden Zeilen `Name` und `Bemerkung` angezeigt. Die Methode `tableView:numberOfRowsInSection:` der Klasse `KundenDetailViewController` gibt deshalb den Wert 2 zurück.

Der Inhalt der Zellen wird wie gewohnt in `tableView:cellForRowAtIndexPath:` definiert:

```
...
   switch (indexPath.row) {
        case 0:
     cell.textLabel.text = @"Name";
     cell.detailTextLabel.text = kunde.name;
     break;
        case 1:
     cell.textLabel.text = @"Bemerkung";
     cell.detailTextLabel.text = kunde.bemerkung;
     cell.detailTextLabel.numberOfLines = 10;
     break;
    }
...
```

Der Zelltyp wird auf `UITableViewCellStyleValue2` festgelegt.

Auf der rechten Seite der Navigation Bar soll wieder der Edit-Button angezeigt werden:

```
- (void)viewDidLoad {
   [super viewDidLoad];
   self.navigationItem.rightBarButtonItem = self.editButtonItem;
   self.title = @"Details";
...
```

Beim Klick auf *Bearbeiten* werden hier aber keine *Löschen*-Icons auf der linken Seite eingeblendet. Die Implementierung der Methode `tableView:editingStyleForRowAtIndexPath:` verhindert das:

```
- (UITableViewCellEditingStyle)tableView:(UITableView *)tableView
   editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
   return UITableViewCellEditingStyleNone;
}
```

Deshalb braucht auf der linken Seite des Views auch kein Platz im *Bearbeiten*-Modus geschaffen zu werden:

```
- (BOOL)tableView:(UITableView *)tableView
shouldIndentWhileEditingRowAtIndexPath:(NSIndexPath *)indexPath {
   return NO;
}
```

Stattdessen bekommt die Zelle im *Bearbeiten*-Modus ein Detail-Icon auf der rechten Seite:

```
cell.editingAccessoryType = UITableViewCellAccessoryDetailDisclosureButton;
```

Außerdem wird dafür gesorgt, dass die Zelle nur im *Bearbeiten*-Modus selektierbar ist und somit auch der Drill-Down nur in diesem Modus ausgeführt werden kann:

```
- (NSIndexPath *)tableView:(UITableView *)tv
   willSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    return (self.editing) ? indexPath : nil;
}
```

Der Drill-Down im Edit-Modus führt zu den Screens zum Bearbeiten der einzelnen Attribute mithilfe von Textkomponenten und Picker. Diese werden im nächsten Kapitel besprochen.

Die in diesem Abschnitt beschriebenen Codierungen, zusammen mit ein paar weiteren Kleinigkeiten, die dem Quellcode der Klasse entnommen werden können, führen zu dem in folgender Abbildung dargestellten View:

Bild 12.23: Detail-View

Es können nach diesem Schritt also vorhandene Datensätze bearbeitet werden (bzw. kann zumindest zu den zu bearbeitenden Details navigiert werden). Das Löschen wurde im Abschnitt »Edit Mode« bereits besprochen. Fehlt noch das Anlegen von neuen Datensätzen, etwa neuen Kunden. Der hierfür zuständige `AddKundeViewController` wird beim Klick auf das Pluszeichen in der Tool Bar der Kundenliste als modaler View eingeblendet. Die Instanzvariable `kunde` bekommt diesmal einen neu erzeugten (aber noch nicht gespeicherten) Kunden zugewiesen. Außerdem wird die Instanz der Klasse `KundenTableViewController` als Delegate gespeichert:

```
- (void)addKunde{

   AddKundeViewController *addViewController = [[AddKundeViewController
     alloc] initWithStyle:UITableViewStyleGrouped];
   addViewController.delegate = self;
   addViewController.kunde = (Kunde *)[NSEntityDescription
     insertNewObjectForEntityForName:@"Kunde"
     inManagedObjectContext:[fetchedResultsController
managedObjectContext]];
   UINavigationController *navController = [[UINavigationController alloc]
     initWithRootViewController:addViewController];
    [self.navigationController
     presentModalViewController:navController animated:YES];
   [addViewController release];
   [navController release];
}
```

Wie dem Header-File von `AddKundeViewController` entnommen werden kann, erbt die Klasse einen Großteil der Funktionalität von `KundenDetailTableViewController`. In der überschriebenen `viewDidLoad`-Methode wird der View der Subklasse mit anderem

Titel und *Abbrechen*- und *Sichern*-Buttons statt der *Back*- und *Bearbeiten*-Buttons der Superklasse versehen:

```
- (void)viewDidLoad {
   [super viewDidLoad];
     self.title = @"Neuer Kunde";

   self.navigationItem.leftBarButtonItem = [[[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemCancel
            target:self action:@selector(cancel:)] autorelease];
   self.navigationItem.rightBarButtonItem = [[[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemSave
            target:self action:@selector(save:)] autorelease];

   self.editing = YES;
}
```

Wird einer der Buttons betätigt, dann wird die weitere Bearbeitung an die Methode `addViewController:didFinishWithSave:` des Delegates (also der Klasse `KundenTableViewController`) weitergereicht:

```
- (IBAction)cancel:(id)sender {
   [delegate addViewController:self didFinishWithSave:NO];
}

- (IBAction)save:(id)sender {
   [delegate addViewController:self didFinishWithSave:YES];
}
```

Dort wird dann der modale View Controller ausgeblendet und im Fall des *Sichern*-Buttons der zuvor zugefügte Datensatz gespeichert.

Auf die Besprechung der analogen Views und Controller für Projekte, Leistungen und Zeiten wird hier wiederum verzichtet. Die Implementierung erfolgt analog.

13 Tab Bar Controller

Tab Bars werden gemäß den *iPhone Human Interface Guidelines* benutzt, um entweder aus verschiedenen Perspektiven auf einen Satz von Daten zu blicken oder aber das Leistungsspektrum der Anwendung in verschiedene Unteraufgaben zu unterteilen. Ein schönes Beispiel für die verschiedenen Perspektiven bietet die iPod-App. Die Tabs zeigen die immer gleichen Musiktitel unter wechselnden Gesichtspunkten wie Interpret, Titel oder Genre. Die Unterteilung der Anwendung in Unteraufgaben meint die bereits besprochene Top-Level-Menü-Funktion.

In Kapitel »User Interface Design« haben wir uns für das *Chronos*-Projekt gegen ein *Tab Bar*-Interface zugunsten einer ständig sichtbaren *Tool Bar* entschieden. Da der *Tab Bar Controller* jedoch eine der zentralen Komponenten für das Zusammenfügen mehrerer Views zu einer Anwendung ist, werden wir uns in diesem Kapitel mit den Möglichkeiten dieses Controllers beschäftigen.

13.1 Funktionsweise des Tab Bar Controllers

Wie auch der Navigation Controller kontrolliert der Tab Bar Controller mehrere Views bzw. deren View Controller. Der Navigation Controller verwaltet die View Controller in Form eines Stapels. Es können durch Methodenaufrufe Controller auf den Stapel gelegt oder vom Stapel genommen werden.

Anders ist die Situation beim Tab Bar Controller. Hier werden alle View Controller nebeneinander in Form einer Liste verwaltet. Jeder View ist durch einen Klick auf den entsprechenden Reiter sofort verfügbar:

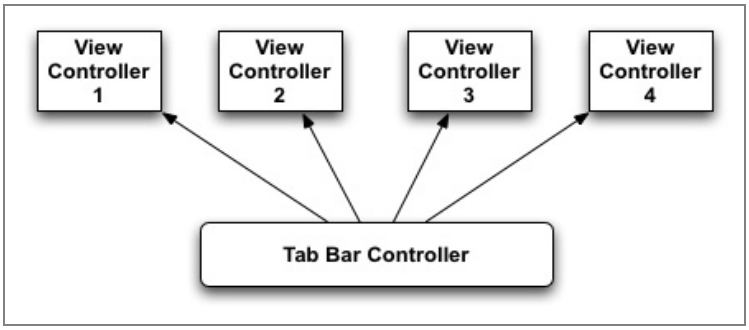


Bild 13.1: Tab Bar Controller & Views

Wird ein Tab ausgewählt, dann wird im Content-Bereich darüber der View des entsprechenden View Controllers angezeigt. Im Gegensatz zum Navigation Controller werden nicht einfach detailliertere, sondern gänzlich andere Informationen angezeigt.

Einer der Tabs muss stets selektiert sein, der Index wird im Tab Bar Controller gespeichert. Um erkennen zu können, welcher Tab gerade aktiv ist, wird die aktuelle Auswahl farblich hervorgehoben.

Die Tab Bar bietet Platz für fünf Items. Werden der Tab Bar mehr Items zugefügt, erhält der fünfte Tab automatisch die Bezeichnung *Mehr*. Ein Klick auf diesen Tab führt zu einer Tabellenansicht mit den verbleibenden Einträgen:

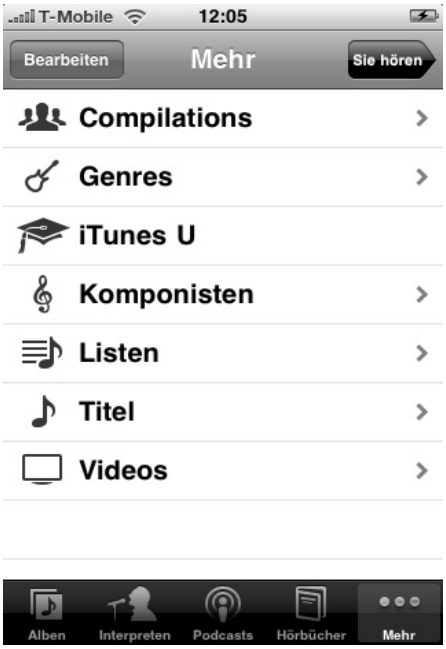


Bild 13.2: Tabelle mit weiteren Optionen

13.2 Erzeugen eines Tab Bar Controllers

Der schnellste Weg, zu einem *Tab Bar*-Interface zu kommen, ist die Nutzung der gleichnamigen Projektvorlage in Xcodes *New Project*-Assistenten. Die Vorlage erzeugt eine einfache Anwendung mit zwei Tabs, die weiter angepasst werden kann:

Bild 13.3: Die Projektvorlage *Tab Bar Application*

Um den Geschehnissen hinter den Kulissen ein wenig auf die Schliche zu kommen, bauen wir ein ähnliches User Interface, beginnen allerdings lediglich mit einer schlichten *Window Based Application*.

Wie Sie aus dem Kapitel »Projektstart« sicherlich noch wissen, enthält die *Window Based Application*-Vorlage keinerlei Views oder View Controller. Die einzige generierte Klasse ist `{Projektname}AppDelegate`.

Ein Doppelklick auf `MainWindow.xib` öffnet das Nib-File im Interface Builder. Hier wird aus der Library ein Tab Bar Controller in das *Document Window* unter *UIWindow* gezogen. Durch den Drag & Drop-Vorgang wird im Nib-File eine Instanz der Klasse erzeugt.

Öffnet man den Tab Bar Controller über das kleine Dreieck links daneben, stellt man fest, dass außer der Tab Bar bereits zwei View Controller enthalten sind:

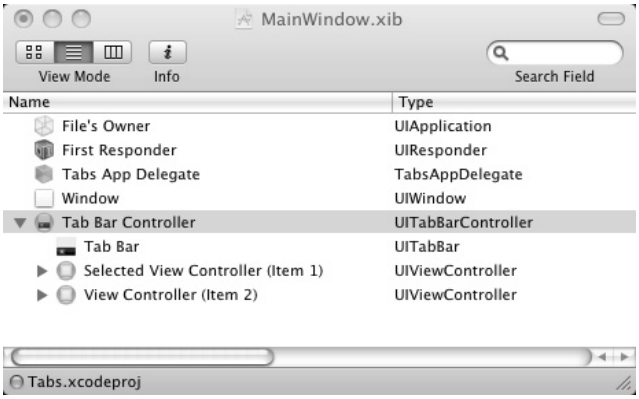


Bild 13.4: Tab Bar Controller im Nib-File

Um das *Tab Bar*-UI zur Anzeige zu bringen, muss es dem Haupt-Window der Anwendung als Subview zugefügt werden. Dazu wird allerdings eine Verbindung des Tab Bar Controllers im Nib-File mit dem Code in Xcode benötigt.

Wir legen also ein Outlet im Header-File der `AppDelegate`-Klasse an:

```
@interface TabsAppDelegate : NSObject <UIApplicationDelegate> {

    IBOutlet UITabBarController *tabBarController;
    UIWindow *window;
}
...
```

Und dann verbinden wir im Interface Builder das Outlet mit dem Tab Bar Controller:

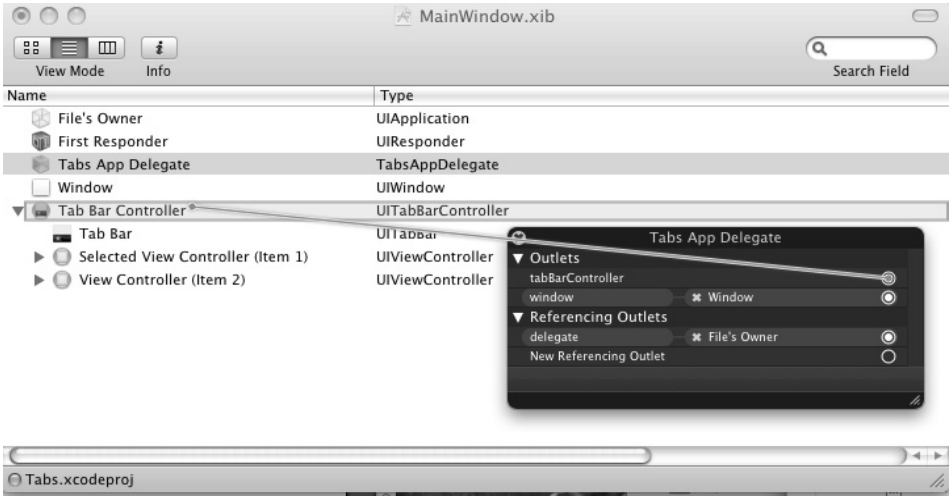


Bild 13.5: Connection für den Tab Bar Controller

Durch das Anlegen der Verbindung wurde eine Referenz auf das Objekt im Nib-File erzeugt.

Als Nächstes wird der aktuelle View des Tab Bar Controllers dem Haupt-Window der App zugefügt. Dies geschieht in der Methode `applicationDidFinishLaunching` der AppDelegate-Klasse:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
   [window addSubview:tabBarController.view];
   [window makeKeyAndVisible];
}
```

Die Tab Bar mit den beiden Default-Items kann nun bereits im Simulator begutachtet werden.

Zur weiteren Zielsetzung orientieren wir uns an der verworfenen Idee eines *Tab Bar*-UI für das *Chronos*-Projekt und fügen zwei weitere View Controller aus der Interface Builder Library zu. Die neuen View Controller müssen innerhalb des Tab Bar Controllers platziert werden. Die Titel der Tab Bar Items können nach Doppelklick direkt in der Designoberfläche angepasst werden:

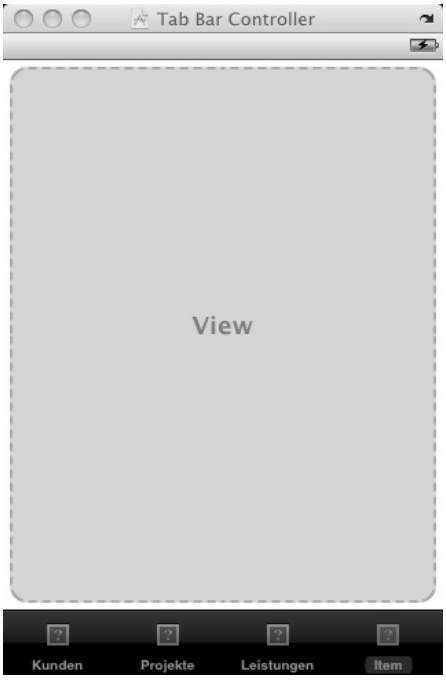


Bild 13.6: Anpassen der Beschriftung

Um die Tab Bar optisch noch etwas anspruchsvoller zu gestalten, werden Icons hinzugefügt. Sind im Projekt Grafiken enthalten, können diese einfach aus dem Media-Bereich der Library per Drag & Drop auf die Tab Bar Items gezogen werden. Falls erst

noch Grafiken hinzugefügt werden müssen, kann das via Kontextmenü im *Groups & Files*-Bereich von Xcode geschehen (*Add > Existing Files...*).

Das Resultat der Bemühungen ist ein durchaus vorzeigbares User Interface, wie die folgende Abbildung zeigt. Man beachte den hervorgehobenen (weil selektierten) Tab *Kunden*:

Bild 13.7: Tab Bar mit Icons

13.3 Tab Bar Controller Delegate

Das Default-Verhalten der Tab Bar genügt den meisten Anwendungsfällen. Besonders die Berücksichtigung von mehr als fünf Items ist beeindruckend. Trotzdem kann es Situationen geben, in denen die Tab Bar an die eigenen Bedürfnisse angepasst werden oder auf Ereignisse reagieren soll. Dazu steht das Protokoll `UITabBarControllerDelegate` zur Verfügung.

Im Protokoll sind fünf optionale Methoden enthalten. Mit ihrer Hilfe kann das Verhalten bei der Selektion von Tabs oder vor oder nach Änderung der Tab-Reihenfolge definiert werden.

Im Einzelnen sind diese Methoden verwendbar:

- `tabBarController:shouldSelectViewController:`
- `tabBarController:didSelectViewController:`
- `tabBarController:willBeginCustomizingViewControllers`
- `tabBarController:willEndCustomizingViewControllers:changed:`
- `tabBarController:didEndCustomizingViewControllers:changed`

Die Vorgehensweise soll anhand eines kleinen Beispiels demonstriert werden.

Zunächst wird eine neue Klasse `TabBarControllerDelegate` mithilfe des *New File*-Assistenten erzeugt. Die Klasse muss das Protokoll `UITabBarControllerDelegate` implementieren:

```
@interface TabBarControllerDelegate :NSObject <UITabBarControllerDelegate> {

}

@end
```

In der Methode `applicationDidFinishLaunching:` wird die `delegate`-Property des Tab Bar Controllers auf eine Instanz der neuen Klasse gesetzt:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
   tabBarControllerDelegate = [[TabBarControllerDelegate alloc]init];
   tabBarController.delegate = tabBarDelegate;
   window addSubview:tabBarController.view];
   [window makeKeyAndVisible];
}
```

Damit der Code funktioniert, muss die Variable `tabBarControllerDelegate` im Interface als Variable und Property definiert und in der Implementierung müssen die `get`- und `set`-Methode mit `synthesize` erzeugt werden (bitte `release` nicht vergessen).

In der Implementierung können nun die gewünschten Methoden aus dem Protokoll implementiert werden. Die folgende Implementierung von `tabBarController:didSelectViewController:` gibt beispielsweise bei jedem Selektieren eines Tabs eine Log-Meldung aus:

```
- (void)tabBarController:(UITabBarController *)tabBarController
   didSelectViewController:(UIViewController *)viewController
{
   NSLog(@"Tab pressed");
...
```

14 Textkomponenten & Picker

In diesem Kapitel sollen die Eingabekomponenten für die Detail-Views von *Chronos* im Vordergrund stehen. Neben den im Beispielprojekt verwendeten Textfeldern und Text Views wird zusätzlich die *Search Bar* vorgestellt. Sie ist eng mit dem Textfeld verwandt und bietet einige zusätzliche suchspezifische Features. Außerdem werden die Picker-Komponenten zur Auswahl von Kunde und Datum besprochen.

14.1 Text Field

Textfelder dienen der Eingabe von eher kurzen Texten. Für die Eingabe steht lediglich eine Zeile zur Verfügung. Da das iPhone über keine echte Tastatur verfügt, wird für die Eingabe der Daten eine virtuelle Tastatur vom unteren Bildschirmrand »hochgefahren«. Die fehlende echte Tastatur ist für Vielschreiber eines der gewichtigsten Argumente gegen das Gerät. Die Eingabe ist zwar im Landscape-Modus der größereren Tasten wegen etwas leichter, trotzdem möchte man damit zugegebenermaßen keine Romane schreiben.

14.1.1 Erzeugung & Konfiguration

Wie alle UI-Komponenten lässt sich ein *Text Field* am einfachsten im Interface Builder per Drag & Drop dem gewünschten View zufügen. Das neue Textfeld kann anschließend über den *Attributes*-Inspektor konfiguriert werden. Properties wie *Placeholder* (ein grauer Text, der einen Hinweis auf die Eingabe gibt) oder *Clear Button* (eine Schaltfläche mit einem Kreuz-Icon zum Löschen des Texts) lassen sich so bequem einstellen.

Die unter *Text Input Traits* im *Attributes*-Inspektor aufgeführten Optionen beeinflussen nicht das Textfeld selbst, sondern beschreiben Merkmale der dazugehörigen Tastatur:

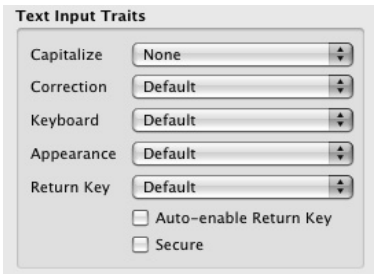


Bild 14.1: Text Input Traits

Besonders interessant sind die Optionen *Keyboard* und *Return Key*. Unter *Keyboard* kann die Art der Tastatur eingestellt werden. Je nach Anwendungsfall kann ausgewählt werden zwischen:

- Default
- ASCII Capable
- Numbers & Punctuation
- URL
- Phone Pad
- Name Phone Pad
- Email Adress

Die Tastatur muss im Normalfall nicht extra eingeblendet werden. Sie erscheint zusammen mit einem blinkenden Cursor automatisch, wenn das Textfeld *First Responder* wird. Dazu muss es vom Benutzer durch einen Tap in das Textfeld hinein als das fokussierte Bedienelement gekennzeichnet werden. Soll ein Textfeld programmatisch als First Responder definiert und dadurch mit Tastatur versehen werden, geschieht das durch folgenden Aufruf:

```
[textField becomeFirstResponder];
```

Wie in der Abbildung zu erkennen ist, kann in den Traits auch der *Return Key* definiert werden. Das ist der Button rechts unten auf der Tastatur, der die Eingabe abschließt und gegebenenfalls eine Aktion unter Verwendung des eingegebenen Texts startet. In der Safari App etwa wird für die URL-Leiste der *Öffnen*-Button und für die Suchleiste ein mit »Google« beschrifteter Button verwendet. Beim Betätigen des Buttons wird die Tastatur nicht automatisch ausgeblendet. Dafür ist eine Delegate-Methode zu implementieren.

14.1.2 Text Field Delegate

Um das Delegate im Code definieren zu können, wird eine Referenz auf das neue Textfeld benötigt. Dafür ist, Sie ahnen es schon, ein Outlet notwendig. Die Klasse, die ein Textfeld repräsentiert, heißt `UITextField`. Auf die detaillierte Besprechung des Anlegens von Instanzvariable, Outlet und Connection verzichten wir an dieser Stelle. Es finden sich im Buch hinreichend Beispiele dafür. Ist das Textfeld im Code verfügbar, wird die `delegate`-Property gesetzt:

```
textField.delegate = self;
```

Alternativ kann die Property `delegate` natürlich auch im Interface Builder mit der passenden Klasse verbunden werden.

Die `delegate`-Klasse ist im Allgemeinen der View Controller des Views, dem das Textfeld hinzugefügt wurde. Er muss das Protokoll `UITextFieldDelegate` implementieren.

Darin enthalten sind einige optionale Methoden, die während des Editiervorgangs aufgerufen werden:

- `textField:shouldChangeCharactersInRange:replacementString:`
- `textFieldDidBeginEditing:`
- `textFieldDidEndEditing:`
- `textFieldShouldBeginEditing:`
- `textFieldShouldClear:`
- `textFieldShouldEndEditing:`
- `textFieldShouldReturn:`

Für das Verhalten des Return-Buttons ist die Methode `textFieldShouldReturn:` zuständig. Möchte man die Tastatur nach einem Tap darauf ausblenden, ist diese Methode zu implementieren und der Status *First Responder* für das Textfeld aufzugeben:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
   [textField  resignFirstResponder];
   return YES;
}
```

Soll anschließend mit dem eingegebenen Text eine Aktion ausgeführt werden, ist die Methode `textFieldDidEndEditing:` des Delegates der geeignete Platz dafür.

14.1.3 Notifications

Sie erinnern sich vielleicht noch an die Diskussion in Kapitel »User Interface Design« zum Thema virtuelle Tastatur und verdeckte UI-Elemente. Der *Notification*-Mechanismus bietet eine weitere Möglichkeit, dem Problem zu begegnen. Im folgenden Code wird ein Observer für eine Notification (Benachrichtigung) namens `UIKeyboardDidShowNotification` definiert:

```
NSNotificationCenter *defaultCenter = [NSNotificationCenter defaultCenter];
[defaultCenter addObserver:self
   selector:@selector(keyboardDidShow:)
   name:UIKeyboardDidShowNotification
   object:nil];
```

Immer wenn das Ereignis eintritt, wird die Methode `keyboardDidShow:` aufgerufen:

```
-(void)keyboardDidShow:(NSNotification *)notification
{
   NSLog(@"Keyboard did show");
}
```

Mit dem analogen Konstrukt für das Verschwinden der Tastatur (`UIKeyboardDidHideNotification`) lässt sich der betroffene View abhängig vom

Vorhandensein der Tastatur in eine Position bringen, in der der gewünschte Inhalt sichtbar ist. Außer den beiden genannten existieren noch die Notifications `UIKeyboardWillShowNotification` und `UIKeyboardWillHideNotification`.

Die grundlegende Verwendung der Textfeld-Komponente sollte damit klar sein. Details zu den nicht angesprochenen Methoden finden sich wie immer in der hervorragenden Referenz von Apple. Im Folgenden soll als praktisches Beispiel die Verwendung in der Zeiterfassung diskutiert werden.

Im vorherigen Kapitel wurde die Implementierung der *Detail*-Views besprochen, die bei einem Tap auf eine der Listen angezeigt werden. Befindet sich der *Detail*-View im *Edit*-Mode, gelangt man durch einen Tap auf eines der Attribute in einen weiteren View zum Bearbeiten der Daten. Die nächste Abbildung zeigt als Beispiel den *Bearbeiten*-View für die Dauer einer erfassten Zeit.

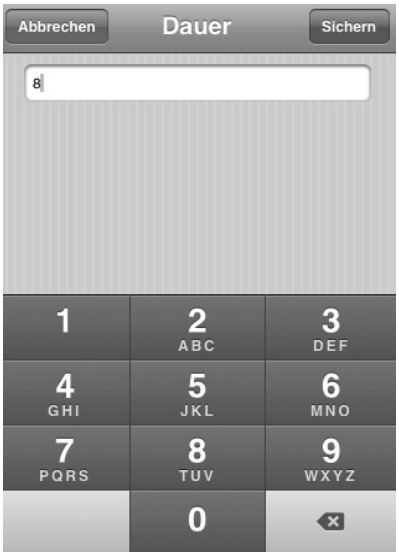


Bild 14.2: *Bearbeiten*-View

Die Realisierung erfolgt in der Klasse `DecimalViewController`, die an ein Codebeispiel von Apple angelehnt wurde. Der Controller wird in `tableView:didSelectRowAtIndexPath:` der Klasse `ZeitenDetailTableViewController` erzeugt. Als Parameter wird der Name des Nib-Files mit dem enthaltenen Textfeld übergeben. Vor der Anzeige des Views werden die Properties `editedObject` (das aktuelle Zeit-Objekt), `editedFieldKey` (das zu bearbeitende Attribut) und `editedFieldName` (der Name des zu bearbeitenden Attributs) gesetzt:

```
...
case 1: {
   DecimalViewController *controller = [[DecimalViewController alloc]
     initWithNibName:@"DecimalView" bundle:nil];

   controller.editedObject = zeit;
```

```
   controller.editedFieldKey = @"dauer";
   controller.editedFieldName = @"Dauer";
   [self.navigationController pushViewController:controller animated:YES];
   [controller release];
} break;
...
```

`EditedFieldName` (im Screenshot oben mit *Dauer* beschriftet) wird in der neuen Klasse `DecimalViewController` als Titel für den *Bearbeiten*-View verwendet:

```
- (void)viewDidLoad {
   self.title = editedFieldName;
```

Der Wert von `zeit.dauer` wird mittels Key Value Coding als Text in das Textfeld gesetzt. Falls `zeit.dauer` den Wert `nil` hat (etwa bei einer neu erfassten Zeit), kommt der Platzhalter zum Zug. Er zeigt dann ebenfalls den Titel, also den Text *Dauer*, an:

```
- (void)viewWillAppear:(BOOL)animated {
   [super viewWillAppear:animated];

   textField.text = [[editedObject valueForKey:editedFieldKey] stringValue];
   textField.placeholder = self.title;
   [textField becomeFirstResponder];
}
```

Die letzte Zeile im Code sorgt dafür, dass das Textfeld bei Erscheinen des Views *First Responder* wird. Die Tastatur wird dadurch eingeblendet, und der Cursor steht im Textfeld.

Wird der *Bearbeiten*-View nach der Eingabe des neuen Wertes mit *Sichern* verlassen, dann wird, wiederum mittels Key Value Coding, der neue Wert in die Variable `dauer` des `zeit`-Objekts geschrieben und der View ausgeblendet (vom Stapel des Navigation Controllers genommen):

```
- (IBAction)save {

   NSDecimalNumber *stundensatz = [[[NSDecimalNumber
     alloc]initWithString:textField.text]autorelease];
    [editedObject setValue:stundensatz forKey:editedFieldKey];

    [self.navigationController popViewControllerAnimated:YES];
}
```

Natürlich hätte ein Controller für das Editieren der `zeit`-Attribute auch etwas einfacher mit einem hart codierten Ansatz entwickelt werden können. Der Vorteil dieser KVC-basierten Variante liegt in der Wiederverwendbarkeit. Der `DecimalViewController` kann für alle Dezimaleingaben von *Chronos* und darüber hinaus sogar noch für andere Projekte verwendet werden.

14.2 Text View

Der *Text View* ist ein UI-Element zum Anzeigen oder auch zur Eingabe von längeren Texten. Im Gegensatz zum Textfeld können in dieser Komponente auch mehrzeilige Texte eingegeben oder verwendet werden. In anderen Programmiersprachen wird ein solches Bedienelement auch als *Textarea* bezeichnet.

Abgesehen von dieser charakteristischen Eigenschaft ähnelt der Text View dem *Text Field* sehr. Auch zu dieser Komponente existiert ein Delegate-Protokoll. Es heißt erwartungsgemäß `UITextViewDelegate` und bietet durch die enthaltenen optionalen Methoden die Möglichkeit, auf verschiedene Editier-Ereignisse zu reagieren.

In unserem Beispielprojekt findet der Text View Verwendung zum Bearbeiten des Bemerkungsfelds, das in allen Detail-Views enthalten ist:

Bild 14.3: Text View in Chronos

Die entsprechende Klasse `TextViewEditController` ist analog zur besprochenen Klasse `DecimalViewController` implementiert und wird hier deshalb nicht weiter diskutiert. Der Text View selbst findet sich im dazugehörigen Nib-File.

14.3 Search Bar

Bei der *Search Bar* handelt es sich im Prinzip um ein erweitertes Textfeld. Auf der linken Seite deutet ein Lupen-Icon auf die Spezialisierung der Komponente hin:

Bild 14.4: Search Bar

Im *Attributes*-Inspektor können per Checkbox der *Bookmark*-Button, der *Cancel*-Button oder die *Scope*-Leiste eingeblendet werden. Die *Scope*-Leiste dient zur Kategorisierung der Suche und ist konfigurierbar. Es lassen sich die Namen der einzelnen Scopes editieren und mit den Plus- bzw. Minus-Buttons sogar Scopes löschen oder hinzufügen. Maximal sind vier Scopes möglich:

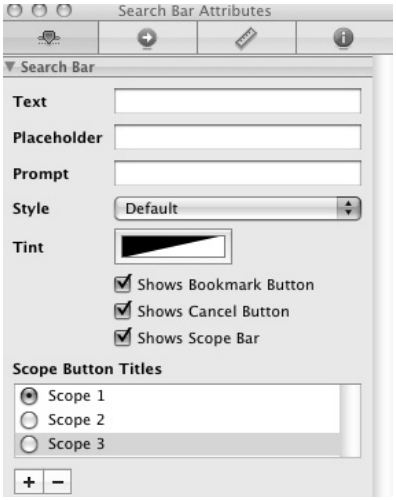


Bild 14.5: Attribute der Search Bar

Um auf die verschiedenen Buttons reagieren zu können, muss auch hier ein Protokoll von der als Delegate definierten Klasse implementiert werden. Der Name des Protokolls ist `UISearchBarDelegate` und es enhält die folgenden optionalen Methoden:

- `searchBar:selectedScopeButtonIndexDidChange:`
- `searchBar:textDidChange:`
- `searchBarBookmarkButtonClicked:`
- `searchBarCancelButtonClicked:`
- `searchBarSearchButtonClicked:`
- `searchBarShouldBeginEditing:`
- `searchBarShouldEndEditing:`
- `searchBarTextDidBeginEditing:`
- `searchBarTextDidEndEditing:`

Auf ein Beispiel wird hier verzichtet. Die Vorgehensweise ist bekannt: Eine Search Bar wird einem View per Drag & Drop hinzugefügt, und die benötigten Buttons werden im *Attributes*-Inspektor aktiviert. Als Delegate der Komponente wird der View Controller des Views gesetzt, auf dem die Search Bar platziert wurde. Dieser muss das Protokoll `UISearchBarDelegate` implementieren und für die aktivierten Buttons die dazugehörigen Methoden aus dem Protokoll zur Verfügung stellen. Die Suche selbst muss natürlich codiert werden.

14.4 Picker View

Der *Picker* ist ohne Zweifel eines der am schönsten animierten Bedienelemente in UIKit. Von der Funktionalität her handelt es sich eigentlich um eine profane Auswahlliste. Diese ist allerdings im Stil einer Walze (wie beispielsweise in Glücksspielautomaten) visualisiert. Sie kann aus mehreren Teilen bestehen. Mit einem »Flick« bringt man die Walze zum Rotieren. Das innerhalb der feststehenden grauen Kennzeichnung (*Selection Indicator*) stehende Element gilt als ausgewählt:

Bild 14.6: Picker View

14.4.1 Erzeugung & Konfiguration

Der Picker wird wie gewohnt aus der Interface Builder Library in einen bestehenden View gezogen. Die Konfigurationsmöglichkeiten im Interface Builder sind begrenzt. Neben den von der Superklasse geerbten Properties lässt sich lediglich auswählen, ob der *Selection Indicator* angezeigt werden soll oder nicht.

Zur Definiton der enthaltenen Daten werden die in den beiden folgenden Abschnitten besprochenen Protokolle `UIPickerViewDelegate` und `UIPickerViewDataSource` verwendet.

Hat man den Picker via Outlet und Instanzvariable im Code zugänglich gemacht, lassen sich einige interessante Methoden der Klasse `UIPickerView` nutzen. Oft benötigt wird `selectRow:inComponent:animated:`, um programmatisch einen der enthaltenen Einträge auszuwählen.

In der Klasse `KundenPickerController` wird so ein bereits für ein Projekt ausgewählter Kunde beim Erscheinen des Picker Views vorselektiert:

```
- (void)viewWillAppear:(BOOL)animated {

   [super viewWillAppear:animated];

   NSError *error;
   if (![[self fetchedResultsController] performFetch:&error]) {
     // Handle the error...
   }

   Projekt *projekt = (Projekt *)editedObject;
   [kundenPicker selectRow:[fetchedResultsController.fetchedObjects
     indexOfObject:projekt.kunde] inComponent:0 animated:YES];
}
```

14.4.2 Picker View Data Source

Das Protokoll `UIPickerViewDataSource` enthält die Methoden `numberOfComponentsIn PickerView:` und `pickerView:numberOfRowsInComponent:`. Mit ersterer wird die Anzahl der Komponenten der Walze, also der Teilwalzen, bestimmt. Die zweite Methode des Protokolls gibt die Anzahl der Einträge pro Komponente wieder.

Die Implementierung der Methoden im `KundenPickerController` der Zeiterfassung sieht wie folgt aus:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)thePickerView {
   return 1;
}

- (NSInteger)pickerView:(UIPickerView *)thePickerView
   numberOfRowsInComponent:(NSInteger)component {
   // alle Kunden plus ein Leereintrag
    return [fetchedResultsController.fetchedObjects count]+1;
}
```

Die Kundenliste besteht nur aus einer Komponente für die Namen der Kunden. Sie enthält so viele Einträge, wie Kunden vorhanden sind. Dazu kommt ein Leereintrag, um einen einmal ausgewählten Kunden auch wieder abwählen zu können.

14.4.3 Picker View Delegate

Das Protokoll `UIPickerViewDelegate` beinhaltet die folgenden Methoden:

- `pickerView:widthForComponent:`
- `pickerView:rowHeightForComponent:`
- `pickerView:didSelectRow:inComponent:`

- `pickerView:titleForRow:forComponent:`
- `pickerView:viewForRow:forComponent:reusingView:`

Die beiden erstgenannten Methoden erlauben das Bestimmen von Höhe und Breite der Komponenten. `PickerView:didSelectRow:inComponent:` wird aufgerufen, wenn ein Eintrag selektiert wird. Die beiden letzteren Methoden dienen schließlich der Definition des Inhalts, also der Einträge. In der Zeiterfassung wird `pickerView:titleForRow:forComponent:` verwendet, um den Picker mit den Namen der Kunden zu füllen:

```
- (NSString *)pickerView:(UIPickerView *)thePickerView
   titleForRow:(NSInteger)row forComponent:(NSInteger)component {

   if (row<[fetchedResultsController.fetchedObjects count])
   {
     Kunde *kunde = [fetchedResultsController.fetchedObjects
        objectAtIndex:row];
     return kunde.name;
   }
   else
     return @"-keiner-";
}
```

Das Ergebnis zeigt die folgende Abbildung:

Bild 14.7: Kundenpicker in der Zeiterfassung

14.5 Date Picker

Der *Date Picker* ist ein spezialisierter Picker zur Datums- und Zeiteingabe. Es stehen verschiedene Modi zur Verfügung, um die Art der Eingabe zu definieren:

- *Date & Time*
- *Date*

- *Time*
- *Timer*

Der Modus kann im Interface Builder über den *Attributes*-Inspektor oder im Quellcode angegeben werden.

Im Buchprojekt wird ein Date Picker zur Eingabe des Datums zu einem Zeiteintrag verwendet. Da die Uhrzeit hier nicht relevant ist, wird der Picker im Modus *Date* betrieben:

Bild 14.8: Date Picker in Chronos

Um den Date Picker auf ein Datum einzustellen, genügt glücklicherweise das Setzen der `date`-Property. Es brauchen also nicht die einzelnen Teilwalzen auf Tag, Monat und Jahr eingestellt werden:

```
NSDate *date = [editedObject valueForKey:editedFieldKey];
if (date == nil) date = [NSDate date];
datePicker.date = date;
```

Abgesehen von diesen Besonderheiten gleicht sich die Implementierung aller Picker Controller, weshalb hier nicht weiter darauf eingegangen wird.

Teil 3 – Erweiterung der Zeiterfassung

Mit Abschluss des letzten Kapitels ist die Zeiterfassung in ihren Grundzügen besprochen. Was nun in Teil 3 folgt, sind mehr oder weniger nette Erweiterungen, die dem Projekt etwas mehr »Glamour« verleihen. Klar, das muss natürlich auch sein. Aber diese Erweiterungen rütteln nicht mehr an der Architektur der Anwendung.

Wenn Sie dem Buch und dem Quellcode bis an diese Stelle auf den Fersen geblieben sind und die Grundkonzepte wie View & Controller, Delegates und Outlets verstanden haben, kommt nun der angenehme Teil.

15 Ortsbestimmung: Core Location

Im Gegensatz zu einem Desktop-Rechner hat der Anwender ein mobiles Gerät wie das iPhone immer in der Tasche. Weiß das Gerät, wo es sich befindet, dann wissen auch die Anwendungen, wo sich der Benutzer befindet. Zusammen mit einem Server, der Informationen über die nähere Umgebung hat, lassen sich damit wirklich faszinierende Anwendungen, sogenannte *Location Based Services*, erstellen.

Als praktisches Beispiel wird die Ortsbestimmung mittels *Core Location* in diesem Kapitel benutzt, um den Projektort in der Zeiterfassung zu speichern. Abhängig davon kann dann beim Erfassen neuer Zeiten das passende Projekt vorbelegt werden. Das funktioniert natürlich nur, wenn die Zeiten auch vor Ort erfasst werden.

15.1 Woher weiß das Gerät, wo es ist?

Die Antwort »GPS« auf diese Frage greift eindeutig zu kurz. Auch ohne freien Blick auf die GPS-Satelliten kennt das Gerät seine Position. Die erste iPhone-Generation und der iPod touch verfügen über keinen GPS-Empfänger und können trotzdem (allerdings mit einer geringeren Genauigkeit) für die Positionsbestimmung verwendet werden. Diese erfolgt über drei verschiedene Wege.

15.1.1 Mobilfunksender

Die erste Möglichkeit zur Positionsbestimmung ist die Lokalisierung der Mobilfunkmasten in der Nähe. Das Gerät sucht in einer Datenbank nach den Masten, die es im Umfeld entdeckt hat. Aus den Standorten der Sender, die in der Datenbank hinterlegt sind, und der Entfernung zu diesen lässt sich die Position des Geräts mit einer Genauigkeit von ein bis zwei Kilometern bestimmen.

15.1.2 WiFi-Netzwerke

Der iPod touch hat keinen GPS-Empfänger und keine Mobilfunk-Funktionalität. Dennoch kann sogar mit diesem Gerät oft die aktuelle Position ermittelt werden. Das Ganze funktioniert ähnlich wie bei der Triangulation mit Mobilfunksendern. Allerdings werden hier die WLANs in der Umgebung zurate gezogen. Womöglich auch Ihres!

Man kann sich gut vorstellen, dass alle Mobilfunkantennen mit ihren Positionen in einer Datenbank zu finden sind. Wie aber kommt denn das private WLAN in eine

Datenbank? Die Antwort klingt einigermaßen verwunderlich, stimmt aber: Unternehmen sammeln, mit oder ohne Unterstützung der WLAN-Besitzer, die Koordinaten aller WLAN-Router. Im Falle des iPhones heißt die Firma Skyhook [URL-SKYHOOK]. Es ist aber davon auszugehen, dass zumindest Google die gleichen Daten erfasst. Skyhook lässt jede Straße mit speziell ausgerüsteten Messfahrzeugen abfahren und speichert zu der jeweiligen Position die MAC-Adresse der Router. Es macht dabei keinen Unterschied, ob das Gerät verschlüsselt ist oder nicht. Zusätzlich bietet die Firma eine Desktop-Software an, die aktive Mitarbeit an der Erstellung der Router-Karte erlaubt. Zumindest in Großstädten, wo die Dichte der Router sehr hoch und das Abfahren der Straßen schnell erledigt ist, kann so die Position inzwischen bis auf wenige Meter bestimmt werden.

15.1.3 GPS

Irgendwo auf dem freien Feld, abseits von allen WLANs und Mobilfunkmasten, hilft nur noch eine Technologie, das *Global Positioning System*, kurz GPS. Ab dem iPhone 3G ist ein Empfänger für die Satellitensignale eingebaut, der die Position bis auf wenige Meter genau bestimmen kann. Seit Apple das entsprechende Verbot aufgehoben hat, finden sich im App Store auch alle bekannten Anbieter von Navigationssoftware, die regen Gebrauch von GPS machen.

15.2 Die Core Location API

Die Schnittstelle für die Ortsbestimmung ist die *Core Location API*. Sie besteht im Wesentlichen aus den drei Klassen `CLLocationManager`, `CLLocationManagerDelegate` und `CLLocation`. Über welchen der drei besprochenen Wege die Position ermittelt wird, bleibt in der API verborgen und braucht den Entwickler nicht zu kümmern.

Vor der Verwendung muss das Framework dem Projekt zugefügt werden. Dazu wird in Xcode das aktuelle Projekt ausgewählt und im Kontextmenü die Option *Add > Existing Frameworks...* gewählt.

15.2.1 Location Manager

Die Klasse `CLLocationManager` ist der Einstiegspunkt in die *Core Location*-API.

Der folgende Code zeigt die typische Verwendung:

```
CLLocationManager *locationManager = [[CLLocationManager alloc]init];

locationManager.desiredAccuracy = kCLLocationAccuracyBest;
locationManager.distanceFilter = 3000; // Meter!

locationManager.delegate = self;

[locationManager startUpdatingLocation];
```

Der *Location Manager* wird zunächst mit `alloc/init` erzeugt. Anschließend werden die Properties `desiredAccuracy` und `distanceFilter` gesetzt.

Die gewünschte Genauigkeit (`desiredAccuracy`) sollte so hoch wie nötig, aber so niedrig wie möglich für den jeweiligen Anwendungsfall gewählt werden. Für eine Pollenflugvorhersage reicht eine Genauigkeit von Kilometern, für ein Navigationssystem dagegen nicht. Hintergrund hierfür ist, dass das Anfordern einer höheren Genauigkeit mehr Energie und Zeit kostet.

Mögliche Werte für `desiredAccurcy` sind:

- `kCLLocationAccuracyBest`
- `kCLLocationAccuracyNearestTenMeters`
- `kCLLocationAccuracyHundredMeters`
- `kCLLocationAccuracyKilometer`
- `kCLLocationAccuracyThreeKilometers`

Ähnlich verhält es sich mit dem Entfernungsfilter (`distanceFilter`), der in Metern angegeben wird. Für viele Anwendungen spielt es keine Rolle, wenn sich der Benutzer ein paar Meter weiter bewegt hat (das Wetter ist zehn Meter weiter genauso schlecht vorhersagbar wie am vorherigen Ort). Ein Update der Position würde lediglich Strom und Zeit kosten und einer Wetter-Anwendung keinen Mehrwert bringen. Deshalb gilt auch für diese Property: So genau wie nötig, aber so ungenau wie möglich.

Nach dem Setzen der Properties bekommt der Location Manager ein Delegate zugewiesen. Dabei handelt es sich um eine Klasse, die das Protokoll `CLLocationManagerDelegate` implementiert. Im einfachsten Fall ist das die gleiche Klasse, in der auch obiges Codestück zu finden ist.

Nun sind alle Vorbereitungen getroffen, es kann mit der Ortsbestimmung begonnen werden. Dazu wird die Methode `startUpdatingLocation` aufgerufen.

Die ermittelte Position kann in einer Callback-Methode des als Nächstes zu besprechenden Protokolls `CLLocationManagerDelegate` abgegriffen werden.

Genügt die neue Position den Ansprüchen, sollte der energieaufwendige Bestimmungsvorgang so schnell wie möglich gestoppt werden. Das wird mit der Methode `stopUpdatingLocation` erledigt.

15.2.2 Location Manager Delegate

Das erwähnte Protokoll `CLLocationManagerDelegate` enthält die Methoden `locationManager:didUpdateToLocation:fromLocation:` und `locationManager:didFailWithError:`. Da die Positionsbestimmung einige Zeit kostet, werden die Callback-Methoden asynchron aufgerufen, wenn ein Hintergrund-Thread die Arbeit erledigt hat.

Innerhalb der Methode `locationManager:didFailWithError:` kann im Fehlerfall auf ein `NSError`-Objekt zugegriffen werden. Es stellt weitere Informationen über den Fehler zur Verfügung. Typische Fehler sind:

- `kCLErrorLocationUnknown`
- `kCLErrorDenied`
- `kCLErrorNetwork`
- `kCLErrorHeadingFailure`

Im folgenden Codefragment ist eine Fehlerauswertung dargestellt:

```
-(void)locationManager:(CLLocationManager *)manager
   didFailWithError:(NSError *)error
{
   NSMutableString *errorString = [[[NSMutableString alloc] init]
autorelease];
   if ([error domain] == kCLErrorDomain) {     // Core Location-Errors

     switch ([error code]) {

       case kCLErrorDenied:
         [errorString appendFormat:@"%@\n",
           NSLocalizedString(@"LocationDenied", nil)];
         break;

       case kCLErrorLocationUnknown:
         [errorString appendFormat:@"%@\n",
           NSLocalizedString(@"LocationUnknown", nil)];
         break;

       default:
         [errorString appendFormat:@"%@ %d\n",
           NSLocalizedString(@"GenericLocationError", nil),
           [error code]];
         break;
     }
   } else {    // Andere Errors
     [errorString appendFormat:@"Error domain: \"%@\"  Error code: %d\n",
       [error domain], [error code]];
     [errorString appendFormat:@"Description: \"%@\"\n",
       [error localizedDescription]];
   }

   // Fehlermeldung anzeigen
    UIAlertView *alert = [[UIAlertView alloc]
     initWithTitle:@"Error"
     message:errorString
```

```
    delegate:self
    cancelButtonTitle:@"OK"
    otherButtonTitles:nil, nil];
   [alert show];
   [alert release];
}
```

Tritt kein Fehler auf, wird die Methode `locationManager:didUpdateToLocation:fromLocation:` mit der neuen Position aufgerufen. Die Ortsdaten sind in einem `CLLocation`-Objekt verpackt, das als Nächstes besprochen wird.

15.2.3 Location

Die Klasse `CLLocation` ist ein Wrapper für das Ergebnis der Ortsbestimmung. Sie enthält die folgenden Informationen in Form von Properties:

- `altitude`
- `coordinate`
- `course`
- `speed`
- `horizontalAccuracy`
- `verticalAccuracy`
- `timestamp`

`altitude` ist die Höhe über dem Meeresspiegel in Metern.

`coordinate` ist vom Typ `CLLocationCoordinate2D` und fasst Breitengrad und Längengrad zusammen.

`course` gibt die Richtung an, in die das Gerät bewegt wird. Die Angabe erfolgt in Gradzahlen (0 ist Norden, Osten 90 usw.).

`speed` ist die Geschwindigkeit des Geräts in m/s.

Die beiden `Accuracy`-Properties machen eine Aussage über den Radius der Ungenauigkeit, die Angabe erfolgt in Metern.

Mithilfe des `timestamp` kann kontrolliert werden, wie aktuell eine Location ist. Da auch gecachte Locations gemeldet werden, können sie damit ausgefiltert werden.

Der folgende Codeabschnitt zeigt die Implementierung der Methode `locationManager:didUpdateToLocation:fromLocation:` für die Zeiterfassung. Beim Anlegen eines neuen Projekts wird die aktuelle Position gespeichert:

```
- (void)locationManager:(CLLocationManager *)manager
   didUpdateToLocation:(CLLocation *)newLocation
        fromLocation:(CLLocation *)oldLocation
{
```

```
   // // Horizontale Koordinaten in Ordnung?
   if (signbit(newLocation.horizontalAccuracy)) {
     NSLog(@"Accuracy ist negativ, Koordinaten nicht verfügbar oder
ungültig");
   } else {
     // Daten aktuell (innerhalb der letzten 10s)?
     NSDate* eventDate = newLocation.timestamp;
     NSTimeInterval howRecent = [eventDate timeIntervalSinceNow];

     if (howRecent < -0.0 && howRecent > -10.0) {
       // Positionsbestimmung stoppen
       [manager stopUpdatingLocation];

       // Zugriff auf die Koordinaten
       projekt.latitude = [NSString stringWithFormat:@"%2.4f",
         newLocation.coordinate.latitude];
       projekt.longitude = [NSString stringWithFormat:@"%2.4f",
         newLocation.coordinate.longitude];

       [self.tableView reloadData];

     }
   }
}
```

Wie im Listing erkennbar ist, wird vor dem Ermitteln der Koordinaten geprüft, ob die Property `horizontalAccuracy` der neuen Position positiv ist. Nur dann sind die Koordinaten auch verwertbar. Man könnte hier noch weiter einschränken und beispielsweise nur Ortsmeldungen mit einer Genauigkeit von 1.000 Metern in die weitere Auswertung einbeziehen.

Außerdem werden alle Positionsmeldungen weggefiltert, die älter als 10 Sekunden sind. Wir bestehen also in dem Beispiel auf einer relativ aktuellen Position. Abhängig vom Anwendungsfall können aber auch ältere gecachte Daten durchaus sinnvoll sein.

15.3 Koordinatenrechnereien

Bei der Erstellung von *Location Based Services* müssen oft Koordinaten in eine Adresse oder umgekehrt eine Adresse in Geokoordinaten umgewandelt werden.

15.3.1 Umwandlung einer Adresse in Koordinaten

Stellen Sie sich eine Anwendung vor, in der ein Vertriebsmitarbeiter über das iPhone die Kunden herausfinden soll, die in der Nähe seiner aktuellen Position wohnen. Aus den Kundenadressen, die auf dem Server vorliegen, müssen nun Geokoordinaten gemacht

und anschließend die Entfernung zwischen aktueller Position und Adresse in Form von Geokoordinaten berechnet werden.

Das iPhone SDK bietet keine Funktionen für die Umwandlung von Adressdaten in Geokoordinaten. Es kann aber für diesen Zweck beispielsweise die Google Maps API benutzt werden. Folgender Aufruf sucht die Geokoordinaten für die Adresse »Unter den Linden 7, 10117 Berlin«.

```
http://maps.google.com/maps/geo?output=xml&q=Unter+den+Linden+7,10117+Berlin
,Deutschland
```

Das Ergebnis in Form von XML sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8" ?>
<kml xmlns="http://earth.google.com/kml/2.0"><Response>
  <name>Unter den Linden 7,10117 Berlin,Deutschland</name>
  <Status>
    <code>200</code>
    <request>geocode</request>
  </Status>
  <Placemark id="p1">
    <address>Unter den Linden 7, 10117 Berlin, Deutschland</address>
    <AddressDetails Accuracy="8"
xmlns="urn:oasis:names:tc:ciq:xsdschema:xAL:2.0"><Country><CountryNameCode>D
E</CountryNameCode><CountryName>Deutschland</CountryName><AdministrativeArea
><AdministrativeAreaName>Berlin</AdministrativeAreaName><SubAdministrativeAr
ea><SubAdministrativeAreaName>Berlin</SubAdministrativeAreaName><Locality><L
ocalityName>Berlin</LocalityName><DependentLocality><DependentLocalityName>M
itte</DependentLocalityName><Thoroughfare><ThoroughfareName>Unter den Linden
7</ThoroughfareName></Thoroughfare><PostalCode><PostalCodeNumber>10117</Post
alCodeNumber></PostalCode></DependentLocality></Locality></SubAdministrative
Area></AdministrativeArea></Country></AddressDetails>
    <ExtendedData>
      <LatLonBox north="52.5204050" south="52.5141098" east="13.3980748"
west="13.3917795" />
    </ExtendedData>
    <Point><coordinates>13.3949287,52.5172503,0</coordinates></Point>
  </Placemark>
</Response></kml>
```

Gibt man die so ermittelten Koordinaten `13.3949287` und `52.5172503,0` in das Suchfeld von Google Maps ein, dann erhält man in der Tat die richtige Adresse in Berlin. Für den oben geschilderten Anwendungsfall ist es natürlich sinnvoll, die Adressdaten bereits auf dem Server in umgewandelter Form vorzuhalten. Das Umwandeln größerer Adressbestände kann über spezialisierte Dienstleister erfolgen.

15.3.2 Umwandlung von Koordinaten in eine Adresse

Auch die Umwandlung in die andere Richtung wird oft benötigt. Denkbar wäre eine Anwendung, die Adressdaten eines Kunden automatisch speichert, wenn vor Ort die Position bestimmt wird. Aus den Geokoordinaten müssen also Straße und Ort ermittelt werden. Hierbei hilft die `MKReverseGeocoder`-Library, die im Map Kit Framework enthalten ist. Sie nutzt intern auch die Google API.

Zur Verwendung der API wird eine Instanz der Klasse `MKReverseGeocoder` erzeugt und mit den gesuchten Koordinaten initialisiert. Die Klasse bekommt anschließend ein Delegate zugewiesen. Das Delegate muss das Protokoll `MKReverseGeocoderDelegate` implementieren, und in den entsprechenden Methoden `reverseGeocoder:didFindPlacemark:` und `reverseGeocoder:didFailWithError:` kann auf die Adresse oder eine Fehlermeldung zugegriffen werden.

Das folgende Codestück demonstriert das Vorgehen:

```
-(void) getAdressFromCoordinate
{
   CLLocationCoordinate2D coordinate;
   coordinate.latitude = 52.5172503;
   coordinate.longitude = 13.3949287;

   MKReverseGeocoder *reverseGeocoder = [[MKReverseGeocoder alloc]
     initWithCoordinate:coordinate];
   reverseGeocoder.delegate = self;
   reverseGeocoder.start;
}

- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder
   didFailWithError:(NSError *)error
{
   NSLog(error.domain);
}

- (void)reverseGeocoder:(MKReverseGeocoder *)geocoder
   didFindPlacemark:(MKPlacemark *)placemark
{
   NSLog(placemark.postalCode);
   NSLog(placemark.locality);
   NSLog(placemark.thoroughfare);
   NSLog(placemark.subThoroughfare);
}
```

Beim Ausprobieren ist zu beachten, dass das Map Kit Framework dem Projekt zugefügt und in den nutzenden Klassen importiert werden muss.

Die Ausgaben der `NSLog`-Befehle für obigen Code sehen folgendermaßen aus:

```
chronos[48269:20b] 10117
chronos[48269:20b] Berlin
chronos[48269:20b] B5
chronos[48269:20b] 7
```

Als Straßennamen erhält man für dieses Beispiel interessanterweise »B5«. Zwar verläuft die gleichnamige Bundesstraße entlang der Straße »Unter den Linden«, aber eigentlich ist das nicht ganz das gewünschte Ergebnis.

15.3.3 Die Entfernungsformel

Die dritte gängige Anforderung für Anwendungen mit Ortsbezug ist die Berechnung der Entfernung zwischen zwei Punkten. Typische Beispiele sind Apps à la »Restaurants in der Nähe«.

Liegen alle Positionen erst einmal in Form von Geokoordinaten vor, lässt sich daraus mit ein wenig sphärischer Trigonometrie die Entfernung berechnen. Die folgende Abbildung illustriert das Problem auf der Erdkugel (R = 6.378,388 km):

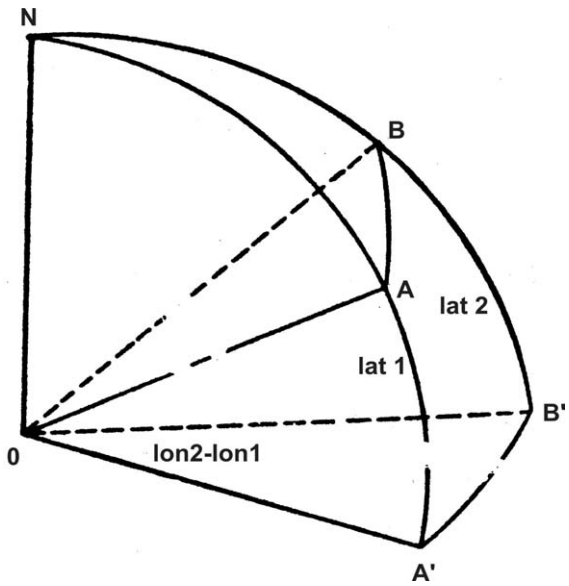


Bild 15.1: Entfernungsberechnung auf der Erdkugel

Zu den beiden gegebenen Punkten A und B wurde noch der Nordpol N dazugenommen. Dadurch erhält man das sphärische Dreieck ABN.

Die Punkte A' und B' liegen auf dem Äquator. Deshalb ist A'N der halbe Meridian und somit gleich 90°, ebenso der andere Meridian B'N. Der Bogen A'A ist die geografische Breite von Punkt A, also *lat1* (das Kürzel *lat* steht hier für *Latitude* = Breite). Der Bogen B'B hat die Größe *lat2*.

Im Dreieck ABN sind also zwei Seiten (AN und BN) bekannt, nämlich gleich `90-lat1` und `90-lat2`; außerdem der von den beiden Meridianen eingeschlossene Winkel, nämlich `lon2 - lon1` (das Kürzel *lon* steht für *Longitude* = Länge), die Differenz der geografischen Längen von B und A.

Sind von einem sphärischen Dreieck zwei Seiten und der eingeschlossene Winkel gegeben, so wird – wie in der ebenen Trigonometrie auch – der Kosinussatz angewandt. In der sphärischen Trigonometrie heißt der:

```
cos c = cos a  cos b  +  sin a  sin b  cos gamma
```

`a` steht für die Seite AN, ist also gleich `90-lat1`; `b` steht für BN und ist gleich `90-lat2`;

`gamma` ist der Winkel zwischen `a` und `b`, also gleich `lon2 - lon1`.

Die gesuchte Seite AB ist das `c` in der Formel. Durch Einsetzen ergibt sich:

```
cos c = cos (90-lat1) * cos (90-lat2)  +  sin (90-lat1) * sin(90-lat2) *
cos (lon2-lon1)
```

Mit `cos(90-x)` gleich `sin(x)` und `sin(90-x)` gleich `cos(x)` folgt weiter:

```
cos c = sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(lon2-lon1)
```

Um daraus nun das `c` zu ermitteln, wird der Arkuskosinus angewendet und das Ganze mit dem Erdradius (6.378,388 km) multipliziert. Außerdem wird verwendet, dass `cos(lon2-lon1) = cos(lon1-lon2)` gilt.

Letztlich resultiert daraus also folgende Gleichung:

```
Entfernung=6378.388
*ARCCOS[SIN(lat1)*SIN(lat2)+COS(lat1)*COS(lat2)*COS(lon1-lon2)];
```

Für die Funktionen SIN (Sinus) und COS (Kosinus) werden alle Koordinaten im Bogenmaß benötigt. 1° entspricht /180 = 0.017453292519943295 8rad. Mit diesem Wert sind Breitengrade und Längengrade zu multiplizieren.

Das folgende Codefragment zeigt die Berechnung in der Methode `locationManager: didUpdateToLocation:fromLocation:` der Klasse `AddZeitViewController`. Beim Erfassen eines neuen Zeiteintrags wird für alle Projekte die Entfernung zur aktuellen Position berechnet. Das nächstliegende Projekt wird vorbelegt, sofern es nicht weiter als 1 km von der aktuellen Position entfernt ist.

```
- (void)locationManager:(CLLocationManager *)manager
   didUpdateToLocation:(CLLocation *)newLocation
        fromLocation:(CLLocation *)oldLocation
{
   if (signbit(newLocation.horizontalAccuracy)) {
     NSLog(@"coordinates not available");
   } else {

     NSDate* eventDate = newLocation.timestamp;
     NSTimeInterval howRecent = [eventDate timeIntervalSinceNow];
```

```
    if (howRecent < -0.0 && howRecent > -10.0) {
      [manager stopUpdatingLocation];


      double currentLocationLongitude =
        newLocation.coordinate.longitude * DEG_TO_RAD;
      double currentLocationLatitude =
        newLocation.coordinate.latitude * DEG_TO_RAD;

      double distanceNearestProjekt = 1000000.0;
      for (Projekt *projekt in fetchedResultsController.fetchedObjects)
      {
        double projektLongitude =
          [projekt.longitude doubleValue] * DEG_TO_RAD;
        double projektLatitude =
          [projekt.latitude doubleValue] * DEG_TO_RAD;

        double distance =  6378.388 *
        acos(sin(currentLocationLatitude) * sin(projektLatitude) +
        cos(currentLocationLatitude) * cos(projektLatitude) *
        cos(projektLongitude - currentLocationLongitude));

        if (distance < distanceNearestProjekt)
        {
          distanceNearestProjekt = distance;
          zeit.projekt = projekt;
        }
      }
      NSLog(@"%f", distanceNearestProjekt);

      // Kein Projekt in der Nähe gefunden
      if (distanceNearestProjekt > 1000)
      {
        zeit.projekt = nil;
      }
      else // Projekt gefunden
      {
        [self.tableView reloadData];
      }
    }
  }
}
```

16 Map Kit

Wie wir im letzten Kapitel gesehen haben, kann man mit den Geokoordinaten allerhand Spielereien betreiben. Noch schöner wäre es allerdings, wenn man die Positionen auch in einer Karte sehen könnte. Bis zum SDK 3.0 musste man dazu *Google Maps* in einem Web View anzeigen. Inzwischen existiert dafür eine eigene Komponente namens *Map View*. Sie ermöglicht als Teil des Map Kit Frameworks den komfortablen Umgang mit Kartendarstellungen, ähnlich denen in der *Maps*-App.

Für die Zeiterfassung wird ein Map View verwendet, um die beim Anlegen des Projekts ermittelte Position anzuzeigen.

16.1 Erzeugung und Konfiguration des Map Views

Ein Map View ist ein View wie jeder andere auch. Die Vorgehensweise zum Erzeugen und Anzeigen der Komponente ist deshalb schon mehrfach besprochen worden und wird nur noch einmal grob skizziert.

Zunächst wird ein neuer View Controller mit leerem Nib-File angelegt. In *Chronos* heißt die Klasse `MapViewController` und das Nib-File `MapView.xib`. Das Nib-File wird geöffnet und ein Map View aus der Interface Builder Library in das *Document*-Fenster gezogen. Wie üblich muss die *Identity*-Property des *File's Owner* auf die Controller-Klasse (in unserem Fall also `MapViewController`) gesetzt und das *View*-Outlet von *File's Owner* mit dem gerade zugefügten Map View verbunden werden.

Im *Attributes*-Inspektor können bei Bedarf noch zusätzlich Konfigurationen vorgenommen werden:

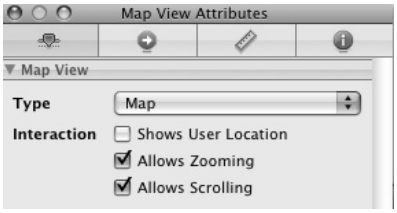


Bild 16.1: Konfigurationsmöglichkeiten des Map Views

Wie bei Google Maps kann als Typ eine Karten-, Satelliten- oder Hybridansicht ausgewählt werden. Mithilfe der Checkbox *Show User Location* wird die aktuelle Position des Geräts auf der Karte angezeigt. Die anderen beiden Optionen erlauben bzw. verhindern das Scrollen und Zoomen des Benutzers in der Karte. Der View ist damit fertiggestellt.

Der Code zum Anzeigen des Views ist ebenfalls bereits wohlbekannt. In *Chronos* erfolgt das Anzeigen aus der Methode `tableView:willSelectRowAtIndexPath:` der Klasse `ProjekteDetailTableViewController:`.

```
...
case 2: {
   MapViewController *controller = [[MapViewController alloc]
     initWithNibName:@"MapView" bundle:nil];
   [self.navigationController pushViewController:controller animated:YES];
   [controller release];
...
```

Die folgende Abbildung zeigt das Resultat der Bemühungen bis zu diesem Punkt:

Bild 16.2: Map View

Das Ergebnis ist schon ganz ermutigend, ohne Zweifel handelt es sich um eine Karte. Allerdings ist die Auflösung – vorsichtig formuliert – etwas grob gewählt, und die beim Anlegen des Projekts ermittelte Position wird nicht markiert. Die Behebung dieser beiden Mängel ist Inhalt der nächsten Abschnitte.

16.2 Regionen

Um den angezeigten Ausschnitt zu bestimmen, besitzt der Map View eine `region`-Property. Die Region wird über einen Mittelpunkt (`center`) und den anzuzeigenden Nord-Süd- bzw. Ost-West-Ausschnitt (`span`) definiert. Bei einem kleinen `span`-Wert wird wenig Entfernung auf der Karte angezeigt, was folglich einen großen Zoom-Level

ergibt (1° entspricht 111 km für `longitudeDelta`). Der folgende Code zeigt das Erzeugen und Setzen der Region in der Zeiterfassung:

```
CLLocationCoordinate2D coordinate;
coordinate.latitude =  [projekt.latitude doubleValue];
coordinate.longitude = [projekt.longitude doubleValue];

MKCoordinateRegion region;
region.center = coordinate;
region.span.longitudeDelta = 0.05f;
region.span.latitudeDelta = 0.05f;
[((MKMapView*)controller.view) setRegion:region animated:YES];
```

Zunächst werden aus den beim Anlegen des Projekts ermittelten Koordinaten `double`-Werte gemacht und damit ein `CLLocationCoordinate2D-Struct` erzeugt. Dieses wird als `center` der Region gesetzt. Für die beiden `span`-Properties wird ein Wert von 0.05 definiert. Wir erhalten damit einen Ausschnitt von einigen Kilometern Breite und Länge auf der Karte. Zu guter Letzt bekommt der Map View die Region gesetzt.

16.3 Annotationen

Beim Anzeigen von Objekten in der Karte helfen Annotationen. Das Protokoll `MKAnnotation` wird von dem Objekt implementiert, das die anzuzeigenden Daten speichert oder zumindest kennt. Dieses Objekt muss auf jeden Fall die `coordinate`-Property mit Daten versorgen. Ergänzend dazu können Titel und Subtitel von der implementierenden Klasse zur Verfügung gestellt werden. Für unsere Projektklasse sieht die `Annotation`-Klasse folgendermaßen aus:

```
#import <MapKit/MapKit.h>

@interface ProjektAnnotation : NSObject <MKAnnotation> {
   CLLocationCoordinate2D coordinate;
   NSString *subtitle;
   NSString *title;
}

@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic,retain) NSString *subtitle;
@property (nonatomic,retain) NSString *title;

-(id)initWithCoordinate:(CLLocationCoordinate2D) coord;

@end
```

```
#import "ProjektAnnotation.h"

@implementation ProjektAnnotation
```

```
@synthesize coordinate, title, subtitle;

-(id)initWithCoordinate:(CLLocationCoordinate2D) coord{
   coordinate = coord;
   return self;
}

- (void)dealloc {
   [title release];
   [subtitle release];
   [super dealloc];
}
@end
```

Die Verwendung dieser neuen Klasse ist recht einfach. Ein `ProjektAnnotation`-Objekt wird mit den Koordinaten erzeugt und bekommt Titel und Subtitel gesetzt. Die Methode `addAnnotation:` des Map Views erlaubt schließlich das Hinzufügen zur Karte:

```
ProjektAnnotation *projektAnnotation = [[ProjektAnnotation alloc]
   initWithCoordinate:coordinate];
projektAnnotation.title = projekt.name;
projektAnnotation.subtitle = projekt.kunde.name;
[((MKMapView *)controller.view) addAnnotation:projektAnnotation];
[projektAnnotation release];
```

Wie die nächste Abbildung zeigt, ist das Ergebnis zufriedenstellend. Die Position des Projekts wird durch eine Stecknadel markiert. Beim Klick auf den roten Nadelkopf werden zusätzlich Projekt- und Kundenname angezeigt.

Zum Anfertigen des Screenshots wurde übrigens keine Reise ins Silicon Valley unternommen. Vielmehr ist es so, dass der Simulator mangels echter Positionsdaten stets das Apple-Hauptquartier in Cupertino als Standort annimmt. Abhilfe kann bei Bedarf beispielsweise das Tool *iSimulate* [URL-ISIMULATE] schaffen, das den Simulator um einige Features erweitert.

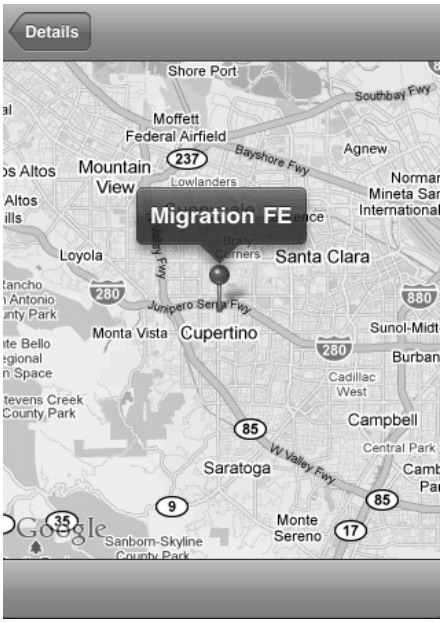


Bild 16.3: Karte mit Pin-Annotation

Als Nächstes soll die Stecknadel gegen eine andere Grafik ausgetauscht werden. Dazu muss man wissen, dass die Visualisierung von Annotationen eigentlich Aufgabe der Klasse `MKAnnotationView` ist. Dass wir die Annotation überhaupt schon zu Gesicht bekommen haben, ist der Verwendung des Default-Views `MKPinAnnotationView` zu verdanken.

Ein Objekt vom Typ `MKAnnotationView` »kennt« über die Property `annotation` sein Annotation-Objekt. Der View wird immer dann angezeigt, wenn sich die von der Annotation gelieferten Geokoordinaten im sichtbaren Bereich der Karte befinden.

Ein eigener View für eine Annotation kann über die Methode `mapView: viewForAnnotation:` des Protokolls `MKMapViewDelegate` definiert werden:

```
- (MKAnnotationView *)mapView:(MKMapView *)lmapView
   viewForAnnotation:(id <MKAnnotation>)annotation {

   MKAnnotationView *view = [[[MKAnnotationView alloc]
     initWithAnnotation:annotation reuseIdentifier:nil]autorelease];
   view.image = [UIImage imageNamed:@"clipboard_small.png"];
   view.canShowCallout = YES;
   return view;
}
```

Im obigen Codebeispiel wird eine Instanz von `MKAnnotationView` erzeugt, mit einer Grafik bestückt und schließlich zurückgegeben. Beachten Sie bitte das `autorelease` beim Erzeugen des Views. Da der View als Rückgabewert dient, besteht keine andere Möglichkeit für die Freigabe. Soll die Grafik im Code gezeichnet werden, kann natürlich

auch eine Subklasse von `MKAnnotationView` erstellt und dort mit der Methode `drawRect:` überschrieben werden.

Damit die Methode überhaupt aufgerufen wird, muss wieder die `delegate`-Property des Map Views auf die das Protokoll implementierende Klasse gesetzt werden. Die folgende Abbildung zeigt das Ergebnis der Programmierarbeiten:

Bild 16.4: Karte mit eigener Annotation

Ein Klick auf die Grafik kann übrigens in der Methode `mapView:annotationView:calloutAccessoryControlTapped:` verarbeitet werden, die ebenfalls Teil des Protokolls `MKMapViewDelegate` ist.

17 Adressbuch

Nach wie vor gehören PIM-(*Personal Information Management*-)Anwendungen wie *Kontakte*, *Kalender* und *Notizen* zu den wichtigsten und am häufigsten benutzten Programmen auf mobilen Geräten. Mit der App *Kontakte* lassen sich Freunde, Bekannte, Kunden usw. zu Gruppen zusammenfassen und Informationen zu den Kontakten, wie Adresse oder Telefonnummer, speichern. Die *Telefon*-Anwendung gestattet den Zugriff auf dieselben Daten, sodass diese nicht doppelt vorgehalten und gepflegt werden müssen. Glücklicherweise ist das kein Privileg der Telefon-App. Jede andere Anwendung kann mithilfe der Frameworks *AddressBookUI* und *AddressBook* die Kontaktdatenbank nutzen. Neben der Möglichkeit, direkt auf die Daten zuzugreifen, existieren komfortable *View Controller* für verschiedene Adressbuch-Aufgaben.

Als praktisches Anwendungsbeispiel bekommt die Zeiterfassung in diesem Kapitel das Feature, einem Projekt einen Ansprechpartner zuzuordnen und diesen bei Bedarf anzuzeigen.

Natürlich müssen vor der Verwendung wieder die beiden Libraries mit dem Projekt verlinkt werden. Dazu wird durch Rechtsklick auf das *Projekt*-Target (*Chronos* im Beispielprojekt) der Info-Dialog geöffnet und mit dem Pluszeichen unter den *Linked Libraries* `AddressBook.framework` und `AddressBookUI.framework` hinzugefügt.

Der Import in die Klassen sieht folgendermaßen aus:

```
#import <AddressBook/AddressBook.h>
#import <AddressBookUI/AddressBookUI.h>
```

17.1 Auswählen von Kontakten

Um einem Projekt einen Kontakt zuordnen zu können, muss dieser zunächst einmal aus der Adressbuchliste aller Kontakte ausgewählt werden. Genau dafür wurde der `ABPeoplePickerNavigationController` geschaffen. Das folgende Codestück in der Klasse `ProjekteDetailTableViewController` zeigt die gewünschte Liste an:

```
ABPeoplePickerNavigationController *controller =
   [[ABPeoplePickerNavigationController alloc]init];
controller.peoplePickerDelegate = self;
[self presentModalViewController:controller animated:YES];
[controller release];
```

Wie man sieht, wird vor dem Anzeigen des Views durch die Methode `presentModalViewController:animated:` ein Delegate (die eigene Klasse) für den Picker definiert. Um als Delegate funktionieren zu können, muss die Klasse

`ProjekteDetailTableViewController` das Protokoll `ABPeoplePickerNavigationControllerDelegate` implementieren, das drei Callback-Methoden enthält.

Die Methode `PeoplePickerNavigationControllerDidCancel:` beschreibt das Verhalten beim Abbruch durch den *Cancel*-Button. Der modale View Controller wird ausgeblendet:

```
-(void)peoplePickerNavigationControllerDidCancel:
   (ABPeoplePickerNavigationController *)peoplePicker
{
   [self dismissModalViewControllerAnimated:YES];
}
```

Die Methode `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:` enthält den Code, der ausgeführt wird, wenn ein Kontakt aus der Liste ausgewählt wird. Über die C-Funktion `ABRecordGetRecordID` wird die `RecordID` der Person ermittelt und mithilfe von *Core Data* in der Datenbank gespeichert. Das ursprüngliche Datenmodell wurde dazu um das Attribut `Projekt.personID` (Typ *Integer 32*) erweitert.

Durch den Rückgabewert `NO` in der Methode wird schließlich verhindert, dass die Person nach der Auswahl angezeigt wird:

```
- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController
*)peoplePicker shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
   projekt.personID = [NSNumber numberWithInt:ABRecordGetRecordID(person)];
   [self dismissModalViewControllerAnimated:YES];
   return NO;
}
```

Oft wird man auch den Namen der Person speichern oder anzeigen wollen. Er kann via `ABRecordCopyCompositeName` ermittelt werden:

```
NSString *name = (NSString *)ABRecordCopyCompositeName(person);
```

Allgemeiner ist die Funktion `ABRecordCopyValue`, mit der alle möglichen Properties kopiert werden können:

```
NSString* name = (NSString *)ABRecordCopyValue(person,
kABPersonFirstNameProperty);
```

Die Konstanten zur Identifikation der Personeneigenschaften sind:

- `kABPersonFirstNameProperty`
- `kABPersonLastNameProperty`
- `kABPersonMiddleNameProperty`
- `kABPersonPrefixProperty`
- `kABPersonSuffixProperty`

- `kABPersonNicknameProperty`
- `kABPersonFirstNamePhoneticProperty`
- `kABPersonLastNamePhoneticProperty`
- `kABPersonMiddleNamePhoneticProperty`
- `kABPersonOrganizationProperty`
- `kABPersonJobTitleProperty`
- `kABPersonDepartmentProperty`
- `kABPersonEmailProperty`
- `kABPersonBirthdayProperty`
- `kABPersonNoteProperty`
- `kABPersonCreationDateProperty`
- `kABPersonModificationDateProperty`

Als letzte Pflichtmethode des Protokolls muss noch `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:` implementiert werden. Die Methode regelt das Verhalten des modalen View Controllers, wenn Eigenschaften des Kontakts, wie etwa eine Telefonnummer, ausgewählt werden. Da in unserem Fall jedoch schon mit der Auswahl des Kontakts der View Controller ausgeblendet wird, spielt die Implementierung keine Rolle.

```
- (BOOL)peoplePickerNavigationController:(ABPeoplePickerNavigationController
*)peoplePicker shouldContinueAfterSelectingPerson:(ABRecordRef)person
property:(ABPropertyID)property
identifier:(ABMultiValueIdentifier)identifier
{
   return NO;
}
```

Der folgende Screenshot zeigt den Picker in Aktion:

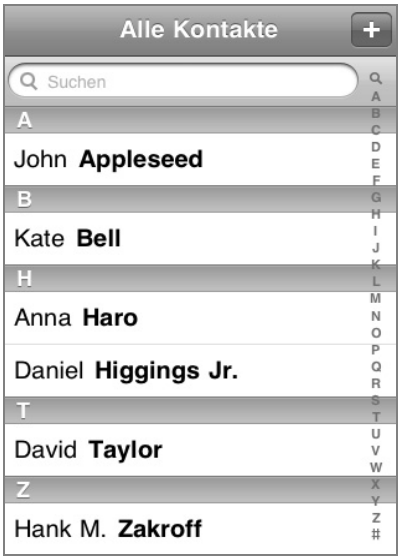


Bild 17.1: Der People Picker

17.2 Anzeigen und Editieren von Kontakten

Die Vorgehensweise, um die Kontaktperson anzuzeigen, ist ganz ähnlich. Der View Controller zum Anzeigen eines Kontakts heißt `ABPersonViewController`.

Dem Controller muss natürlich die Person mitgeteilt werden, die angezeigt werden soll. Zum Auslesen von Personendaten aus dem Adressbuch dient die Funktion `ABAddressBookGetPersonWithRecordID`. Sie wiederum benötigt neben der `RecordID` der Person noch ein `ABAdressBook`, das durch die Funktion `ABAddressBookCreate` erzeugt wird.

Der View Controller bekommt ebenfalls ein Delegate gesetzt. Schließlich wird er nach der Erzeugung mittels der Methode `pushViewController:animated:` zur Anzeige gebracht:

```
ABPersonViewController *controller = [[ABPersonViewController alloc]init];
ABAddressBookRef addressBook = ABAddressBookCreate();
controller.displayedPerson = ABAddressBookGetPersonWithRecordID(addressBook,
[projekt.personID intValue]);
controller.personViewDelegate = self;
[self.navigationController pushViewController:controller animated:YES];
[controller release];
CFRelease(addressBook);
```

Auch hier muss wegen des Delegates ein Protokoll implementiert werden. Es heißt `ABPersonViewControllerDelegate` und beinhaltet die Methode `personViewController:shouldPerformDefaultActionForPerson:property:identifier:`.

Sie gibt an, ob die Default-Aktion, wie beispielsweise Anrufen beim Klick auf eine Telefonnummer, ausgeführt werden soll:

```
- (BOOL)personViewController:(ABPersonViewController *)personViewController
shouldPerformDefaultActionForPerson:(ABRecordRef)person
property:(ABPropertyID)property
identifier:(ABMultiValueIdentifier)identifier
{
   return YES;
}
```

Die folgende Abbildung zeigt die zugeordnete Kontaktperson in *Chronos*:

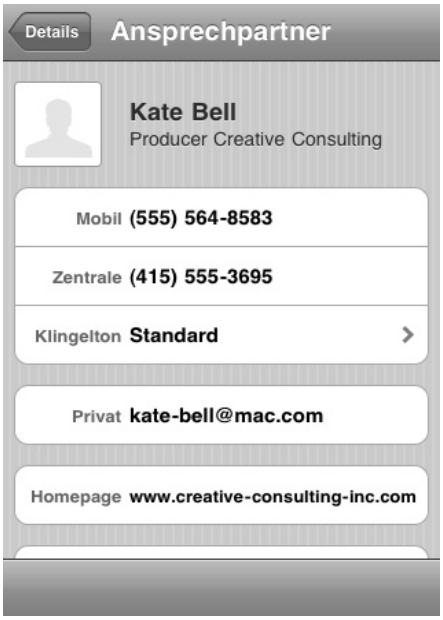


Bild 17.2: Zugeordnete Kontaktperson

17.3 Anlegen von neuen Kontakten

Das Framework *AdressBookUI* enthält noch zwei weitere Controller für das Erzeugen neuer Personendaten. Die Verwendung ist ähnlich wie bei den vorgestellten Klassen.

Der erste Controller nennt sich `ABNewPersonViewController`. Er zeigt den aus der *Kontakte*-App bekannten View zum Neuanlegen von Kontakten an:

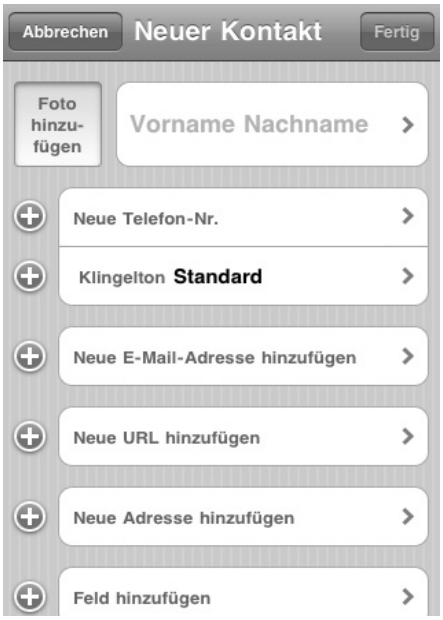


Bild 17.3: Anlegen neuer Kontakte

Das Delegate des Controllers muss das Protokoll `ABNewPersonViewControllerDelegate` implementieren. Dieses enthält als einzige Methode `newPersonViewController:didCompleteWithNewPerson:`. Sie wird aufgerufen, wenn der Benutzer das Anlegen des neuen Kontakts mit *Speichern* oder *Abbrechen* beendet. In dieser Methode ist lediglich der View Controller mit `dismissModalViewControllerAnimated:` auszublenden, das Speichern des neuen Kontakts übernimmt das Framework.

Der andere Controller heißt `ABUnknownPersonViewController`. Er erlaubt das Hinzufügen von neuen Daten zu bestehenden Kontakten oder das Neuanlegen von Kontakten mit den neuen Daten. Sie kennen den View vermutlich aus der *Mail*-App. Wird dort auf eine E-Mail-Adresse getapt, zu der noch kein Kontakt existiert, wird folgender View angezeigt:

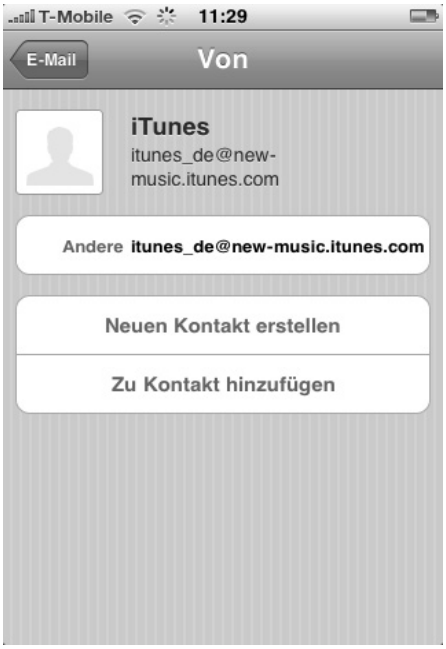


Bild 17.4: Unknown Person View Controller

Das dazugehörige Protokoll heißt `ABUnknownPersonViewControllerDelegate` und enthält die Methoden `unknownPersonViewController:didResolveToPerson:` sowie `unknownPersonViewController:shouldPerformDefaultActionForPerson:property:identifier:`.

Kontakte können mittels der im Framework *AdressBook* enthaltenen C-Funktionen auch ohne die beschriebenen View Controller angelegt und bearbeitet werden. Das folgende Listing zeigt, wie ein neuer Kontakt im Adressbuch angelegt wird:

```
ABAddressBookRef addressBook = ABAddressBookCreate();
ABRecordRef person = ABPersonCreate();
CFErrorRef error = NULL;
ABRecordSetValue(person, kABPersonFirstNameProperty,
          (CFStringRef)@"Peter", &error);
ABRecordSetValue(person, kABPersonLastNameProperty,
          (CFStringRef)@"Lustig", &error);
if(NULL != error) {
   NSLog(@"Error");
}
ABAddressBookAddRecord(addressBook, person, &error);
ABAddressBookSave(addressBook, &error);
CFRelease(person);
CFRelease(addressBook);
```

Zunächst wird eine Referenz auf das Adressbuch und eine neue Person erzeugt. Die Person bekommt Vor- und Nachnamen gesetzt und wird anschließend dem Adressbuch zugefügt. Beachten Sie bitte das Casting auf den in der Funktion `ABRecordSetValue`

benötigten *Core Foundation*-String. Das Adressbuch wird danach gespeichert, und die Objekte werden wieder freigegeben.

Zum Löschen von Einträgen wird die Funktion `ABAddressBookRemoveRecord` verwendet. Weitere Funktionen sind dem Referenzmaterial zu entnehmen.

18 Zugriff aufs Internet

Die Innovationen, die uns das iPhone und nachfolgende Generationen mobiler Endgeräte bringen, sind nicht in erster Linie die spannenden Spiele oder die iPod-Funktionalität. Das wirklich Aufregende ist die ständige Verbindung mit dem Internet und die Applikationen, die daraus ihren Nutzen ziehen. In Verbindung mit der Kamera und den Geokoordinaten lassen sich Anwendungen schreiben, die unser Leben verändern werden. Alle benötigten Informationen werden ständig verfügbar sein, egal wo wir uns gerade befinden. Dem lang ersehnten mobilen Internet wurde mit dem iPhone zum Durchbruch verholfen.

In diesem Kapitel steht das Thema Internet im Vordergrund. Nach den Grundlagen wie dem Versenden von Mails oder dem Anzeigen von Webseiten auf dem Gerät werden anspruchsvollere Themen wie der Datenaustausch mit einem Server über XML und JSON (JavaScript Object Notation) besprochen.

18.1 Mails versenden

Vor dem Erscheinen des SDK 3.0 konnten Anwendungen E-Mails nur dann ohne größeren Aufwand versenden, wenn aus der Anwendung heraus die *Mail*-App aufgerufen wurde. Das eigene Programm wurde dabei beendet und musste wieder neu gestartet werden. Leider blieb dabei die Benutzbarkeit der Anwendung auf der Strecke.

Mit dem SDK 3.0 hat Apple ein Einsehen gehabt und endlich das lang ersehnte In-App-Mail-Feature spendiert.

Für das Versenden von Mails aus der Anwendung heraus wird das Framework *MessageUI* benötigt. Ein Rechtsklick auf das Target der App und die anschließende Auswahl der Option *Get Info* öffnet den Dialog zum Hinzufügen der Bibliothek:

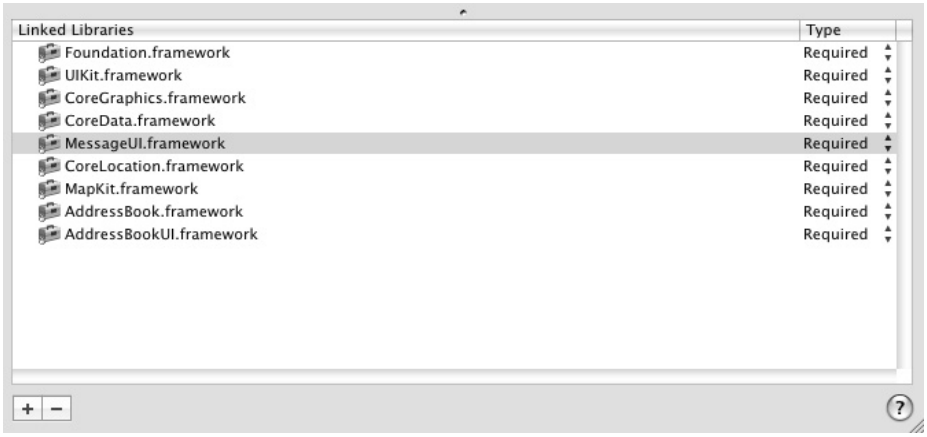


Bild 18.1: Hinzufügen des Frameworks *MessageUI*

Vor der Verwendung muss das Framework wie üblich importiert werden:

```
#import <MessageUI/MessageUI.h>
```

Herzstück der Bibliothek ist die Klasse `MFMailComposeViewController`, ein View Controller, der den View zum Versenden der Mail anzeigt:

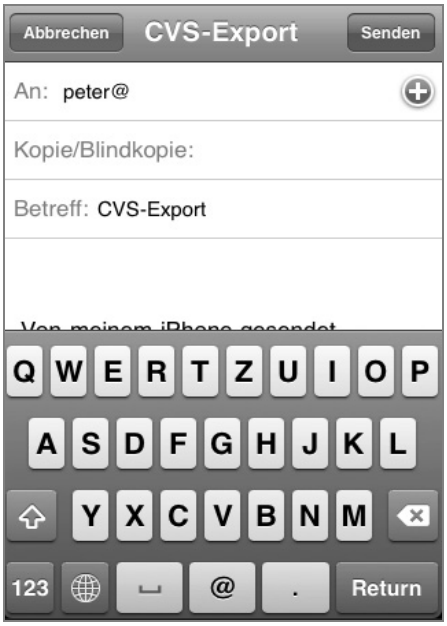


Bild 18.2: In-App-E-Mail

Der dazugehörige Code ist überschaubar. Ein `MFMailComposeViewController` wird mit `alloc/init` erzeugt, bekommt Empfänger, Betreff und Inhalt gesetzt und wird danach, am besten als modaler View Controller, angezeigt:

```
MFMailComposeViewController *controller = [[MFMailComposeViewController
   alloc]init];
[controller setToRecipients:[NSArray arrayWithObject:@"peter@work.xy"]];
[controller setSubject:@"CVS-Report"];
[controller setMessageBody:@"text" isHTML:NO];
[self presentModalViewController:controller animated:YES];
[controller release];
```

Der Code kann zum Beispiel als Reaktion auf einen Tool-Bar-Button-Klick in der zugehörigen Action-Methode verwendet werden.

Irgendwie muss der modale View nach dem Versenden der Mail oder nach einem Tap auf den *Abbrechen*-Button wieder ausgeblendet werden können. Dazu ist noch eine Callback-Methode aus dem Protokoll `MFMailComposeViewControllerDelegate` zu implementieren:

```
- (void)mailComposeController:(MFMailComposeViewController*)controller
didFinishWithResult:(MFMailComposeResult)result error:(NSError*)error {
   [self becomeFirstResponder];
   [self dismissModalViewControllerAnimated:YES];
}
```

Damit die Methode aufgerufen wird, müssen das Protokoll `MFMailComposeView ControllerDelegate` der Liste der Protokolle im Header-File hinzugefügt und das Delegate nach dem Erzeugen des Controllers gesetzt werden:

```
MFMailComposeViewController *controller = [[MFMailComposeViewController
   alloc]init];
controller.mailComposeDelegate = self;
[controller setToRecipients:[NSArray arrayWithObject:@"peter@work.nil"]];
...
```

18.2 Web View

Eine der häufigsten Anforderungen aus dem Themenkomplex Internet ist sicherlich das Anzeigen einer Webseite innerhalb der App. Das kann eine Seite sein, die real auf einem Webserver liegt und somit unabhängig von der Anwendung geändert werden kann. Diese Variante bietet die Möglichkeit, Daten in der App ohne aufwendiges neues Deployment zu aktualisieren.

Alternativ dazu kann der HTML-Code aber auch in einem Verzeichnis der Anwendung abgelegt und somit als statischer Teil des Programms, ähnlich einem im Interface Builder gestalteten View, ausgeliefert werden. Diese Möglichkeit wird oft für *Info/About*-Views verwendet. Es können leicht Links, zum Beispiel auf eine Hilfeseite, untergebracht werden. Im Gegensatz zu den Komponenten *Text View* und *Text Field*, die nur einen

Schriftstil pro Container erlauben, kann in einer Webseite die ganze Palette der HTML-Gestaltungsmöglichkeiten genutzt werden. Texte mit verschiedenen Schriftgrößen, Schriftarten oder Farben lassen sich also am einfachsten als Webseite realisieren.

Zum Anzeigen der Seiten steht eine eigene Komponente namens *Web View* in der Interface Builder Library zur Verfügung. In der Zeiterfassung wird die Komponente für einen *About*-Dialog genutzt.

Als Erstes wird ein View Controller für den Web View benötigt. Die Subklasse von `UIViewController` erhält den Namen `InfoViewController` und wird wie gewohnt mit dem *New File*-Assistenten angelegt. Der Controller bekommt eine Outlet-Instanzvariable vom Typ `UIWebView` mitsamt Property hinzugefügt:

```
@interface InfoViewController : UIViewController {
   IBOutlet UIWebView *webView;
}
@property (nonatomic, retain) IBOutlet UIWebView *webView;
@end
```

In der Implementierung der Klasse erfolgt wie üblich das Erzeugen der Accessor-Methoden mit `synthesize` und das Freigeben des Web Views in der `dealloc`-Methode:

```
@implementation InfoViewController

@synthesize webView;

- (void)dealloc {
   [webView release];
   [super dealloc];
}

@end
```

Danach wird ein leeres Nib-File erzeugt (*File > New File > User Interface > Empty XIB*). Das File wird im Interface Builder geöffnet und bekommt aus der Library einen Web View ins Document Window geschoben. *File's Owner* des Nib-Files soll der neue View Controller sein, er wird dazu im Identity-Inspektor als *Identity Class* gesetzt.

Abschließend werden die `view-` und `webView`-Outlets über eine Connection mit dem Web View verbunden. Die Arbeiten im Interface Builder sind damit beendet.

Im Code soll der neue View als modaler View angezeigt werden. Die Tool Bar erhält zu diesem Zweck einen neuen Info-Button (in der Methode `viewDidLoad` der Klasse `MenuTableViewController`).

```
[self setToolbarItems:[NSArray arrayWithObjects:
   [[[UIBarButtonItem alloc] initWithImage:
   [UIImage imageNamed:@"info_small.png"]
   style:UIBarButtonItemStylePlain
   target:self
   action:@selector(showInfo)]autorelease],
      nil]];}
```

Beim Betätigen des Buttons wird die Methode `showInfo` aufgerufen. Hier wird der View Controller instanziiert. Der Name des neuen Nib-Files wird als Parameter übergeben. Damit der Info-Dialog wieder verlassen werden kann, wird der Controller in einen Navigation Controller mit *Cancel*-Button eingebettet. Dieser wird dann als modaler View Controller angezeigt:

```
- (void)showInfo {
   UIViewController *controller = [[InfoViewController alloc]
     initWithNibName:@"InfoView" bundle:nil];
   UINavigationController *navController = [[UINavigationController alloc]
     initWithRootViewController:controller];
   controller.navigationItem.leftBarButtonItem = [[[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemCancel
   target:self action:@selector(cancelInfo)] autorelease];
    [self.navigationController presentModalViewController:navController
     animated:YES];
   [controller release];
   [navController release];
}
```

Als Letztes muss der Web View nun mitgeteilt bekommen, welche Seite er anzeigen soll. Dazu wird die Methode `viewDidLoad` im `InfoViewController` überschrieben. Zu Testzwecken nutzen wir zunächst einmal eine bestehende Seite. Aus einem String mit der Adresse wird ein `NSURL`-Objekt und aus diesem wiederum ein `NSURLRequest` erzeugt. Die Methode `loadRequest`, aufgerufen mit dem Request als Parameter, bringt schließlich die Seite zur Anzeige.

```
- (void)viewDidLoad {
   NSString *address = @"http://developer.apple.com/iphone";
   NSURL *url = [NSURL URLWithString:address];
   NSURLRequest *request = [NSURLRequest requestWithURL:url];
   [webView loadRequest:request];
}
```

Die folgende Abbildung zeigt den Web View mitsamt Navigation Bar im Simulator:

Bild 18.3: Web View in der App

Zur Anzeige eines HTML-Files aus der *Resources*-Gruppe des Projekts kann stattdessen folgender Code verwendet werden:

```
NSString *filePath = [[NSBundle mainBundle] pathForResource:@"InfoViewHtml"
ofType:@"html"];
NSData *htmlData = [NSData dataWithContentsOfFile:filePath];
if (htmlData) {
   [webView loadData:htmlData MIMEType:@"text/html"
     textEncodingName:@"UTF-8" baseURL:nil];
}
```

Bei Bedarf kann über folgende Callback-Methoden der Klasse `UIWebViewDelegate` auf Ereignisse beim Laden der Webseite reagiert werden:

- `webView:didFailLoadWithError:`
- `webView:shouldStartLoadWithRequest:navigationType:`
- `webViewDidFinishLoad:`
- `webViewDidStartLoad:`

18.3 Request & Response

Bei Daten aus dem Internet muss es sich nicht um Webseiten handeln, die in einem Web View angezeigt werden sollen. Es können Daten in einem beliebigen Format sein, die von der Anwendung geladen, aufbereitet und zum Beispiel in einem Table View zur Anzeige gebracht werden.

Zum Erstellen eines Lade-Requests müssen, ähnlich wie beim *Core Data*-Stack, eine ganze Menge Klassen zusammenarbeiten. Man spricht hier auch vom Cocoa *URL Loading System*. Aus einem `NSString`-Objekt wird ein `NSURL`-Objekt und aus diesem wiederum ein `NSMutableURLRequest`-Objekt erzeugt. Die Instanz des Requests wird dann mithilfe der Klasse `NSURLConnection` versendet. Als Antwort werden die geladenen Daten in Form einer `NSData`-Instanz ein Objekt vom Typ `NSURLResponse` sowie gegebenenfalls ein Fehler-Objekt erhalten. Die einzelnen Schritte werden nun ausführlich betrachtet.

18.3.1 URL

Die Klasse `NSURL` kapselt eine URL (*Uniform Resource Locator*), also eine Ressource im Netzwerk. In diesem Kapitel ist damit eine Internetadresse gemeint. Es könnte sich aber auch um Dateien in Verzeichnissen handeln. Grundsätzlich können die folgenden Protokolle verwendet werden: `http://`, `https://`, `ftp://` und `file://`.

Zum Erstellen einer URL steht eine ganze Reihe von Methoden zur Verfügung. Am einfachsten ist das Erzeugen der URL mithilfe eines Strings über die Klassenmethode `URLWithString:`:

```
NSURL *url = [NSURL URLWithString:@"http://www.heise.de"];
```

18.3.2 URL Request

Mithilfe der URL kann nun ein Lade-Request erstellt werden:

```
NSURLRequest *request = [NSURLRequest requestWithURL:url];
```

Der `NSURLRequest` kapselt die URL und einige weitere Eigenschaften, wie zum Beispiel Caching-Verhalten und Timeout.

Um mehr Einflussmöglichkeiten auf den Request zu haben, kann statt `NSURLRequest` die Subklasse `NSMutableURLRequest` verwendet werden. Sie erlaubt beispielsweise die Änderung der HTTP-Methode, um Daten mittels `POST` an einen Server zu verschicken:

```
[myRequest setValue:@"text/xml" forHTTPHeaderField:@"Content-type"];
[myRequest setHTTPMethod:@"POST"];
[myRequest setHTTPBody:myData];
```

18.3.3 URL Connection

Der Request kann nun mithilfe der Klasse `NSURLConnection` versendet werden. Dieser Vorgang kann entweder synchron oder asynchron erfolgen. Die asynchrone Variante ist zu bevorzugen, weil beim synchronen Laden die Anwendung so lange blockiert ist, bis der Request komplett abgeschlossen ist. Bei größeren Dateien bzw. längeren Netzwerkzugriffen hat der Benutzer dadurch den Eindruck, dass die Anwendung abgestürzt ist. Außerdem können beim asynchronen Zugriff die ersten Daten schon angezeigt werden, bevor der Request vom Server komplett abgearbeitet wurde. Das führt zu einem deutlich besseren Bediengefühl.

Synchrones Versenden

Zum synchronen Versenden des Requests dient die Klassenmethode `sendSynchronousRequest:returningResponse:error:`. Neben dem Request bekommt die Methode `Out`-Parameter für das Response-Objekt vom Server sowie ein `Error`-Objekt für eine Fehlermeldung. Die eigentlich interessierenden geladenen Daten werden in einem Objekt vom Typ `NSData` erhalten:

```
NSURLResponse *response;
NSError *error;
NSData *data = [NSURLConnection sendSynchronousRequest:request
   returningResponse:&response error:&error];

   if ([ error code]) {
     NSLog(@"%@ %d %@", [ error domain], [ error code],
     [error localizedDescription]);
     // evtl. Alert View anzeigen
   }
...
```

Zum Testen, ob der Request erfolgreich war, kann man das Ergebnis während der Entwicklung ausgeben:

```
NSLog([[NSString alloc]initWithData:data
encoding:NSISOLatin1StringEncoding]);
```

Im Normalfall werden die Daten natürlich in irgendeiner Form weiterverarbeitet.

Asynchrones Versenden

Der bessere, aber auch aufwendigere Weg zum Versenden von Requests ist die asynchrone Variante. Hier werden beim Eintreffen der Ergebnisse vom Server Callback-Methoden aufgerufen. Zum Laden der Daten wird der Request der Methode `initWithRequest:delegate:` übergeben. Das Delegate ist die Klasse, in der die Callback-Methoden implementiert sind.

Nach dem Erstellen des `NSURLConnection`-Objekts kann dieses gleich wieder freigegeben werden:

```
[UIApplication sharedApplication].networkActivityIndicatorVisible = YES;
NSURLConnection *connection = [[NSURLConnection alloc]
   initWithRequest:request delegate:self];
[connection release];
```

Die Callback-Methode zum Abgreifen der Daten heißt `connection:didReceiveData:`. Die Methode wird nicht nur einmal aufgerufen, sondern immer dann, wenn der Server ein neues Stück der Antwort bereitstellt. Es kann also, falls die gesamte Nachricht in einem Objekt benötigt wird, notwendig sein, die Teilstücke mithilfe der Klasse `NSMutableData` aneinanderzuhängen:

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData
*)data
{
   NSLog([[NSString alloc] initWithData:data
     encoding:NSISOLatin1StringEncoding]);
   [receivedData appendData:data];   // Objekt vom Typ NSMutable Data
}
```

Das Ende des Requests wird durch Aufruf der Methode `connectionDidFinishLoading:` signalisiert. Die Methode ist ein guter Platz, um etwa einen während des Ladens angezeigten Aktivitätsindikator zu stoppen oder die Verarbeitung der gesammelten Daten anzustoßen:

```
- (void) connectionDidFinishLoading: (NSURLConnection*) connection {
   [UIApplication sharedApplication].networkActivityIndicatorVisible = NO;
}
```

Eventuell auftretende Fehler werden in `connection:didFailWithError:` behandelt:

```
- (void)connection:(NSURLConnection *)connection
   didFailWithError:(NSError *)error
{
   if ([ error code]) {
     NSLog(@"%@ %d %@", [ error domain], [ error code],
     [error localizedDescription]);
   // evtl. Alert View anzeigen
   }
}
```

Eine weitere interessante Callback-Methode ist `connection:didReceiveResponse:`. Das übergebene `NSURLResponse`-Objekt enthält wichtige Informationen wie zum Beispiel die erwartete Größe (`expectedContentLength`), den MIME-Typ oder den Zeichensatz der Antwort:

```
- (void)connection:(NSURLConnection *)connection
   didReceiveResponse:(NSURLResponse *)response
{
   NSLog(@"expectedContentLength: %i", [response expectedContentLength]);
   NSLog(@"MIME-Typ: %@", [response MIMEType]);
   NSLog(@"textEncodingName: %@", [response textEncodingName]);
}
```

18.4 Austauschformate: XML und JSON

Für das Datenformat zur Kommunikation mit einem Server im Internet existieren einige Alternativen. Das bewährte, aber voluminöse XML hat in den letzten Jahren überall dort, wo es auf Geschwindigkeit und kleine Datenmengen ankommt, mit JSON *(JavaScript Object Notation)* Konkurrenz bekommen.

18.4.1 XML

XML ist das bekannteste Austauschformat. Zu den Vorteilen gehören die gute Lesbarkeit sowie die große Menge an verfügbaren Parsern und anderen hilfreichen Bibliotheken in so ziemlich allen gängigen Programmiersprachen. Die Unterstützung von XML in iPhone OS ist nicht ganz so umfangreich wie auf dem Mac, aber brauchbar.

Parsen von XML

Zum Parsen (sprich: Entschlüsseln) von XML steht ein eventbasierter Parser in Form der Klasse `NSXMLParser` zur Verfügung. Um sie zu verwenden, wird eine Instanz der Klasse erzeugt und mit den zu parsenden XML-Daten initialisiert. Der Parser bekommt dann ein Delegate gesetzt, im einfachsten Fall die eigene Klasse. Schließlich wird das Parsen mit der Methode `parse` gestartet:

```
NSXMLParser *parser = [[NSXMLParser alloc] initWithData:data];
[parser setDelegate:self];
[parser parse];
[parser release];
```

Während des Parsing-Vorgangs werden die im Delegate implementierten Callback-Methoden aufgerufen. Wie bei eventbasierten Parsern üblich, wird dort das Auftreten bestimmter Ereignisse gemeldet, und bei Interesse kann darauf reagiert werden. Die Liste der verwertbaren Ereignisse ist lang. Sie kann in der Dokumentation der Klasse eingesehen werden.

Die wichtigsten Methoden sind in der folgenden Tabelle wiedergegeben:

Delegate-Methode	*Wird aufgerufen, wenn ...*
`parserDidStartDocument:`	der Parser mit dem Parsen des Dokuments beginnt.
`parser:didStartElement:namespaceURI:qualifiedName:attributes:`	der Start-Tag für ein Element gefunden wird.
`parser:didEndElement:namespaceURI:qualifiedName:`	der Ende-Tag für ein Element gefunden wird.
`parser:foundCharacters:`	ein Textfragment des aktuellen Elements zur Verfügung steht.
`parserDidEndDocument:`	der Parser mit dem Parsen des Dokuments fertig ist.
`parser:parseErrorOccurred:`	Fehler während des Parsens auftreten.

Da anders als bei DOM-Parsern nicht das gesamte XML-Dokument als Baumstruktur im Speicher vorliegt, muss der Inhalt interessierender Elemente während des Parsens mithilfe der Methoden in Variablen übertragen werden. Betrachten wir ein konkretes Beispiel. Das XML für drei Zeiteinträge von *Chronos* könnte folgendermaßen aussehen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<zeiten>
   <zeit>
     <datum>17.01.2010</datum>
     <dauer>2</dauer>
     <projekt>Aragon></projekt>
     <leistung>Coaching</leistung>
   </zeit>
   <zeit>
     <datum>18.01.2010</datum>
     <dauer>2.5</dauer>
     <projekt>Migration F></projekt>
     <leistung>Entwicklung</leistung>
   </zeit>
   <zeit>
     <datum>18.01.2010</datum>
     <dauer>6</dauer>
     <projekt>Migration F></projekt>
     <leistung>Entwicklung</leistung>
   </zeit>
</zeiten>
```

Um nun aus dem XML ein Array mit drei `Zeit`-Objekten und den entsprechenden Daten zu erstellen, müssen die Callback-Methoden implementiert werden. Wir beschränken uns aus Platzgründen auf die Auswertung der Dauer. In `parserDidStartDocument:` wird ein

neues Array zur Aufnahme der Zeiten instanziiert. Die Methode wird einmalig aufgerufen:

```
- (void)parserDidStartDocument:(NSXMLParser *)parser
{
   array = [[NSMutableArray alloc] init];
}
```

In der für die Element-Detektion zuständigen Methode wird ein neues `Zeit`-Objekt angelegt, wenn der `zeit`-Tag registriert wird. Ebenso wird ein neuer String erzeugt, wenn der Tag `<dauer>` gefunden wird. Das `dauer`-Tag findet sich im XML immer nach `<zeit>`, sodass immer schon ein `Zeit`-Objekt zur Aufnahme der Dauer bereitsteht:

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString
*)qualifiedName attributes:(NSDictionary *)attributeDict
{
   if ( [elementName isEqualToString:@"zeit"] ) {
     zeit = [[Zeit alloc]init];
   }

   if ( [elementName isEqualToString:@"dauer"] ) {
     currentStringValue = [[NSMutableString alloc]initWithCapacity:100];
     return;
   }
}
```

In `parser:foundCharacters:` werden die ganzen String-Fragmente, aus denen der Inhalt eines Elements bestehen kann, zusammengefügt. Für das Element `<projekt>Migration F</projekt>` würde die Methode zweimal aufgerufen werden.

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
   if (currentStringValue != nil)
   {
     [currentStringValue appendString:string];
   }
}
```

Wenn der Ende-Tag des Elements `dauer` auftaucht, wird der Inhalt von `currentStringValue` (also die addierten Fragmente) als Dauer für das zuvor angelegte `Zeit`-Objekt gesetzt. Danach wird die Variable freigegeben. Wird der Tag `</zeit>` gefunden, kann das `Zeit`-Objekt dem Array zugefügt und ebenfalls mit `release` freigegeben werden:

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString *)elementName
namespaceURI:(NSString *)namespaceURI qualifiedName:(NSString *)qName
{
   if([elementName isEqualToString:@"dauer"]){
     zeit.dauer = currentStringValue;
```

```
    [currentStringValue release];
    currentStringValue = nil;
    return;
  }
  if([elementName isEqualToString:@"zeit"]){
    [array addObject:zeit];
    [zeit release];
    return;
  }
}
```

Am Ende des Parse-Prozesses liegen die drei `Zeit`-Objekte im Array vor und können weiterverarbeitet werden:

```
- (void)parserDidEndDocument:(NSXMLParser *)parser
{
  for(Zeit *aZeit in array)
  {
    NSLog(aZeit.dauer);
  }
  [array release];
}
```

Eventuell auftretende Fehler werden in `parser:parseErrorOccurred:` gemeldet:

```
- (void)parser:(NSXMLParser *)parser parseErrorOccurred:(NSError
*)parseError
{
  NSLog(@"%@ %d %@", [ parseError domain], [ parseError code],
  [parseError localizedDescription]);
}
```

Dieser zugegebenermaßen sehr einfache Code soll nur das Prinzip verdeutlichen. Insbesondere für kompliziertere XML-Strukturen, womöglich mit gleichen Tag-Namen an unterschiedlichen Stellen im Baum, wäre noch einige Arbeit nötig.

18.4.2 JSON

Gerade die für Menschen gute Lesbarkeit von XML, die eine Folge des Textformats und gut gewählter Elementnamen ist, bläht die reine Datenmenge etwas auf. Den reinen Nutzdaten steht oft die gleiche Menge an strukturierenden Metadaten gegenüber. Für viele Anwendungsbereiche ist das kein Problem, und die größere Datenmenge wird wegen der Vorteile gerne in Kauf genommen.

Für das iPhone ist allerdings gerade der schnelle Datenaustausch mit dem Internet einer der wichtigsten Erfolgsfaktoren. Apps mit langen Ladezeiten stören die Bedienbarkeit und werden vom Benutzer nicht lange toleriert. Deshalb muss die Datenmenge möglichst klein gehalten werden.

Mit *JSON* [URL-JSON] hat in den letzten Jahren eine etwas leichtgewichtigere Alternative zu XML an Beliebtheit gewonnen. Es handelt sich dabei um ein ursprünglich aus JavaScript stammendes Textfomat, in dem die Nutzdaten in Form von *Key/Value*-Paaren und Arrays gespeichert werden. *Key* und *Value* stehen in geschweiften Klammern und werden durch einen Doppelpunkt getrennt. Bei mehreren Paaren erfolgt die Trennung durch ein Komma:

```
{"key1":value1,
"key2":value2, ...}
```

Die einzelnen Elemente eines Arrays stehen in eckigen Klammern, durch Komma getrennt:

```
[value1, value2]
```

Für eine weitergehende Beschreibung sei auf die zahlreichen Tutorials im Internet oder die oben genannte Quelle im Internet verwiesen.

Generieren von JSON

Für die Zeiterfassung wird das JSON-Format benutzt, um auf Wunsch die erfassten Zeiten an einen Server zu übertragen und daraus einen Stundenzettel im PDF-Format zu erzeugen. Die Generierung des PDFs erfolgt also nicht auf dem iPhone, sondern wird an den Server delegiert. Auf dem in Java realisierten Server stehen neben einer leistungsfähigeren Hardware bewährte Bibliotheken zur PDF-Erzeugung zur Verfügung. Folgende Schritte sind zu realisieren:

1) Umwandeln der erfassten Zeiten in das JSON-Format
2) Übertragen der Daten an den Server
3) Umwandeln der JSON-Daten in Objekte
4) Erzeugen des PDF aus den Objekten
5) Versand des PDF als Mail

Die Schritte 3 bis 5 erfolgen auf dem Server und sollen hier nicht tiefergehend besprochen werden. Eine einfache Implementierung in Java könnte zum Beispiel Java Servlets, die *iText*-Bibliothek [URL-ITEXT] sowie die *Java Mail*-API nutzen.

Für das Umwandeln der erfassten Zeiten in das JSON-Format wird das Open-Source-Framework *TouchJSON* [URL-TOUCHJSON] genutzt. Es handelt sich dabei um einen in Objective-C implementierten Parser und Generator für JSON.

Nach dem Download und dem Entpacken der Sourcen von TouchJSON wird der Inhalt des Ordners *source* dem eigenen Projekt zugefügt. Das kann in Xcode nach einem Rechtsklick auf die Projektdatei über das Kontextmenü (*Add > Existing Files...*) erfolgen. Wurden die Sourcen noch nicht in den eigenen Projektordner kopiert, muss dabei die Option *Copy items into destination group's folder* selektiert werden.

Zur Erzeugung des JSON-Codes dient die Klasse `CJSONSerializer`. Sie bekommt das umzuwandelnde Objekt übergeben. In unserem Fall handelt es sich dabei um ein Array

von Zeiten. Die folgende Methode `prepareJSON:` zeigt, wie aus den Daten der Datenbank ein String im JSON-Format erzeugt wird:

```
-(NSString *) prepareJSON:(NSArray *)fetchedObjects
{
   NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
   [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
   [dateFormatter setTimeStyle:NSDateFormatterNoStyle];

   NSMutableArray *arrayDict = [[NSMutableArray alloc] init];
   for(Zeit *zeit in fetchedObjects)
   {
     NSArray *keys = [NSArray arrayWithObjects:@"projekt", @"leistung",
       @"datum", @"dauer", nil];
     NSArray *objects = [NSArray arrayWithObjects:zeit.projekt.name,
       zeit.leistung.name, [dateFormatter stringFromDate:zeit.datum],
       [zeit.dauer stringValue], nil];

     NSDictionary *dictionary = [NSDictionary dictionaryWithObjects:objects
       forKeys:keys];
     [arrayDict addObject:dictionary];
   }
   NSString *jsonString = [[CJSONSerializer serializer]
serializeObject:arrayDict];
   [arrayDict release];
   [dateFormatter release];

   return jsonString;
}
```

Zunächst wird ein `NSDateFormatter` zum Formatieren des Datums erzeugt. Der Stil der Datumsformatierung ist `NSDateFormatterMediumStyle`. Monatsnamen werden im Gegensatz zur Verwendung von `NSDateFormatterLongStyle` also nicht ausgeschrieben. Auf eine Zeitangabe wird komplett verzichtet. Anschließend wird ein leeres Array erzeugt. Dieses bekommt der `CJSONSerializer` später übergeben. In einer Schleife über alle der Methode übergebenen Zeiteinträge wird dem Array für jeden Schleifendurchgang ein `NSDictionary`-Objekt hinzugefügt. Das Dictionary wird mithilfe der Klassenmethode `dictionaryWithObjects:forKeys:` aus einem Key-Array und einem Value-Array erzeugt. Die Keys sind die Namen der Eigenschaften des Zeit-Objekts wie `datum`, `dauer`, `leistung` usw. Die Values sind die entsprechenden Werte. Die folgende Abbildung zeigt den Aufbau der Datenstruktur:

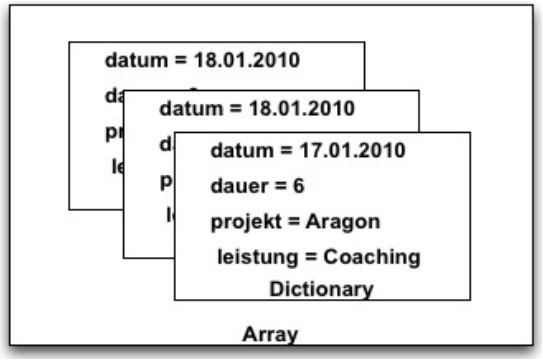


Bild 18.4: Datenstruktur für die Umwandlung in JSON

Das Array wird schließlich der Methode `serializeObject:` einer `CJSONSerializer`-Instanz als Parameter übergeben.

Der dadurch generierte JSON-String sieht für die drei Zeiteinträge folgendermaßen aus:

```
[
{"datum":"17.01.2010","leistung":"Coaching","projekt":"Aragon","dauer":"8"},
{"datum":"18.01.2010","leistung":"Entwicklung","projekt":"Migration
F","dauer":"2.5"},
{"datum":"18.01.2010","leistung":"Entwicklung","projekt":"Migration
F","dauer":"6"} ]
```

Die eckigen Klammern enthalten ein Array mit drei Einträgen. Die Einträge sind durch Kommas getrennt und stehen in geschweiften Klammern. Innerhalb der geschweiften Klammern stehen vier Key/Value-Paare, wiederum durch Kommas getrennt.

Der JSON-String ist übrigens bei nahezu gleichem Informationsgehalt mit ca. 250 Zeichen deutlich kleiner als sein XML-Gegenstück mit ca. 450 Zeichen.

Die erfassten Zeiten sind damit in das JSON-Format umgewandelt und können an den Server übertragen werden. Der folgende Codeabschnitt zeigt die dafür zuständige Methode `sendData:`.

```
-(void)sendData:(NSString *)jsonString
{

   NSString *url = [[NSString
stringWithFormat:@"http://dk.appspot.com/server"]
     stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
   NSURL *myURL = [NSURL URLWithString:url];
   NSMutableURLRequest *myRequest = [NSMutableURLRequest
     requestWithURL:myURL];
   NSData *postData = [jsonString dataUsingEncoding:NSASCIIStringEncoding
     allowLossyConversion:NO];
   [myRequest setHTTPMethod:@"POST"];
   myRequest setHTTPBody:postData];
```

```
   [myRequest setValue:@"application/json" forHTTPHeaderField:@"Content-
Type"];
   NSString* requestDataLengthString = [[NSString alloc]
initWithFormat:@"%d",
     [postData length]];
   [myRequest setValue:requestDataLengthString
     forHTTPHeaderField:@"Content-Length"];
   [requestDataLengthString release];
   [[NSURLConnection alloc] initWithRequest:myRequest delegate:self
     startImmediately:YES];
}
```

Große Teile dieses Listings sind bereits bekannt. So wird zunächst aus einem String mit der Server-URL ein `NSURL`-Objekt und aus diesem wiederum ein `NSMutableURLRequest` erzeugt. Dem Request wird die HTTP-Methode `POST` gesetzt, und der zuvor in ein `NSData`-Objekt umgewandelte JSON-String wird in den HTTP-Body geschrieben. Anschließend werden noch die Header-Informationen `Content-Type` und `Content-Length` gesetzt und der Request mit der Methode `initWithRequest:delegate:startImmediately:` abgeschickt.

Da das generierte PDF auf dem iPhone nicht sonderlich weiterhilft, wird die Antwort des Servers nicht als Request übermittelt. Stattdessen wird eine E-Mail mit dem Timesheet als Anhang versendet, die der Benutzer dann wahlweise auch auf einem Desktop-Rechner öffnen und sie dort zum Beispiel ausdrucken kann.

Parsen von JSON

Für den umgekehrten Weg, also die Umwandlung von JSON in Objekte, kann bei Bedarf der in der Bibliothek enthaltene `CJSONDeserializer` benutzt werden:

```
NSData *jsonData = [jsonString dataUsingEncoding:NSUTF8StringEncoding];
NSError *error;
NSArray *array = [[CJSONDeserializer deserializer] deserialize:jsonData
error:&error];

for(NSDictionary *zeitDict in array)
{
   NSLog([zeitDict objectForKey:@"projekt"]);
}
```

Die Dokumentation auf der Homepage des Open-Source-Projekts *TouchJSON* enthält dazu weitere Informationen.

19 File I/O

Neben der SQLite-Datenbank bietet sich natürlich das Dateisystem an, um Daten persistent zu speichern. Insbesondere für kleinere Datenmengen kann das ein sinnvoller Weg sein, weil der Zugriff relativ einfach ist. Für größere Datenmengen, bei deren Verwendung die Performance eine wichtige Rolle spielt, ist die Datenbank zu bevorzugen.

Aus Sicherheitsgründen hat der Entwickler keinen Zugang zum gesamten Dateisystem des Geräts. Wäre das der Fall, könnten Daten anderer Anwendungen (beispielsweise die Kontakte) ausgelesen und womöglich ins Internet übertragen werden. Aus diesem Grund steht nur das *Home*-Verzeichnis der App für Lese- und Schreibvorgänge zur Verfügung. Man spricht hier auch vom *Sandbox*-Prinzip.

19.1 Verzeichnisstruktur

Das *Home*-Verzeichnis jeder Anwendung enthält vier Unterordner, die spezifische Aufgaben erfüllen. Am einfachsten können Sie sich die Ordner ansehen, die für den Start auf dem Simulator zusammengestellt wurden. Sie sind unter *~/Library/Application Support/iPhone Simulator/User/Applications* zu finden. In diesem Verzeichnis liegen ein oder mehrere Verzeichnisse mit einem hexadezimalen String als Bezeichnung (`etwa 0EA3422E-65AB-4F2B-93D7-3D1A0C366318`). Jeder Ordner steht für eine App. In dem Ordner liegen die besagten vier Unterordner:

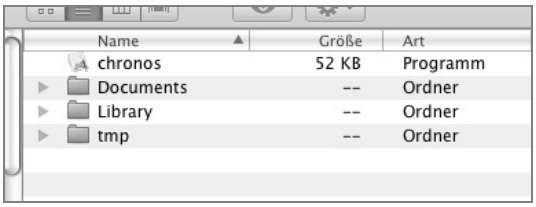


Bild 19.1: Home-Verzeichnis

Beim Abspeichern von Daten in den Ordnern sollte man sich sehr genau überlegen, um welche Art von Daten es sich jeweils handelt und in welchen Ordnern sie deshalb abgelegt werden müssen. Der Grund dafür ist die unterschiedliche Behandlung der Verzeichnisse bei der Wiederherstellung und beim Backup.

19.1.1 tmp

Wie gewohnt dient das *tmp*-Verzeichnis dem vorübergehenden Speichern von Daten. Der Ordner ist für Daten gedacht, die für die Dauer eines Laufs der App gespeichert

werden müssen. Dateien in diesem Ordner werden deshalb vom Backup nicht gesichert. Der Entwickler ist zuständig für das regelmäßige Aufräumen im Verzeichnis. Es erfolgt also kein automatisches Löschen der temporären Inhalte.

19.1.2 Documents

In das Verzeichnis *Documents* sollten alle dauerhaft benötigen Daten wie Textdateien, Bilder oder Datenbanken gespeichert werden. Dies ist also das wichtigste Verzeichnis, wenn es um nicht temporäre Daten geht. Der Ordner wird durch das Backup gesichert.

19.1.3 {App-Name}.app

Im Verzeichnis, das den gleichen Namen wie die Anwendung trägt, liegt die App selber. Der Ordner hat die nicht angezeigte Endung *.app*, weshalb er nicht geöffnet werden kann. Im Finder kann jedoch über das Kontextmenü und die Option *Paketinhalt anzeigen* der Inhalt des Ordners betrachtet werden. In ihm finden sich der Code und die zugehörigen Ressourcen wie beispielsweise Nib-Files. Zur Ausführungszeit der App werden Daten aus diesem Verzeichnis gelesen. Sie dürfen auf keinen Fall in das Verzeichnis schreiben, weil dadurch die Signierung der Anwendung zerstört wird und diese dann nicht mehr startet.

19.1.4 Library/Preferences

Hier landen die Einstellungen, die über die *Preferences*-API (beispielsweise mit der im nächsten Kapitel zu besprechenden Klasse `NSUserDefaults`) gemacht werden. In das Verzeichnis sollte nicht einfach mit anderen Mitteln wie etwa dem später zu besprechenden `NSFileManager` geschrieben werden. Das Verzeichnis wird beim Backup gesichert.

19.1.5 Library/Caches

Das Verzeichnis *Library/Caches* dient, wie der Name schon verrät, als Cache für die App. Der Ordner muss vom Entwickler explizit angelegt werden. Hier können Daten wie zum Beispiel Bilder abgelegt werden, die über die Dauer der App-Benutzung hinaus gespeichert werden sollen. Innerhalb des App-Codes wird typischerweise nachgesehen, ob die betroffene Datei schon im Cache-Ordner vorhanden ist. Ist das der Fall, wird sie verwendet. Wenn nicht, muss sie beschafft (zum Beispiel aus dem Internet geladen) werden. Da das Verzeichnis von iTunes bei einer Wiederherstellung gelöscht wird, müssen diese Daten allerdings auf jeden Fall wiederbeschafft werden können. Der Ordner wird beim Backup nicht berücksichtigt.

19.2 Pfade

Um eine Datei laden oder speichern zu können, wird zunächst einmal der Pfad zum gewünschten Verzeichnis benötigt.

Den Pfad zum Home-Verzeichnis erhält man über die Funktion `NSHomeDirectory`:

```
NSString *homeDirectory = NSHomeDirectory();
```

Den Pfad zum App-Verzeichnis kann man einfach mit der Methode `resourcePath` der Klasse `NSBundle` erhalten:

```
NSString *appDir = [[NSBundle mainBundle] resourcePath];
```

Die anderen Pfade sind mithilfe der Funktion `NSSearchPathForDirectoriesInDomains` ermittelbar. Der erste Parameter der Funktion ist eine Konstante, die für das gewünschte Verzeichnis steht:

Konstante	*Verzeichnis*
`NSDocumentDirectory`	*‹Application_Home›/Documents*
`NSCachesDirectory`	*‹Application_Home›/Library/Caches*
`NSApplicationSupportDirectory`	*‹Application_Home›/Library/Application Support*

Auf dem iPhone ist der Pfad immer das erste Element im zurückgegebenen Array.

Beispiel:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
   NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
```

Beim Arbeiten mit Pfaden ist die Methode `stringByAppendingPathComponent:` der Klasse `NSString` sehr hilfreich. Sie erweitert den Pfad um das übergebene Fragment, wobei sie, falls nötig, den Pfad-Separator (`/`) einfügt. So kann mithilfe des Pfades zum Home-Verzeichnis ebenfalls leicht auf die anderen Ordner zugegriffen werden:

```
NSString *tmpDirectory = [homeDirectory
   stringByAppendingPathComponent:@"tmp"];
```

19.3 File Manager

Eine leistungsfähige Schnittstelle zum Dateisystem ist die Klasse `NSFileManager`. Sie bietet sehr viele Methoden zum Verschieben, Kopieren, Löschen, Erstellen und Auslesen von Dateien und Verzeichnissen.

Man erhält die Singleton-Instanz des File Managers über die Klassenmethode `defaultManager`:

```
NSFileManager *fileManager = [NSFileManager defaultManager];
```

Im Folgenden wird eine Auswahl von wichtigen Operationen anhand von Beispielen besprochen. Die komplette Auflistung aller Methoden ist der Klassendokumentation zu entnehmen.

Schreiben

Das folgende Beispiel demonstriert das Schreiben eines Text-Files ins Dateisystem. Zuerst wird mit den bereits besprochenen Methoden der Speicherpfad im *Documents*-Directory definiert. Das Anlegen der Datei erledigt die Methode `createFileAtPath:contents:attributes:`. Mit `NSFileHandle fileHandleForWritingAtPath:` wird eine Referenz auf die neue Datei erhalten, mit deren Hilfe schließlich der Inhalt geschrieben wird (via `writeData:`).

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSString *documentsPath = [NSSearchPathForDirectoriesInDomains
   (NSDocumentDirectory, NSUserDomainMask, YES) objectAtIndex:0];
NSString *filePath = [documentsPath stringByAppendingPathComponent:
   @"myFile.txt"];

NSFileHandle *savedFile = nil;
if([fileManager createFileAtPath:filePath contents:nil attributes:nil])
{
   savedFile = [[NSFileHandle fileHandleForWritingAtPath:filePath] retain];
}

NSString *string = @"Text";
[savedFile writeData:[string dataUsingEncoding:NSUTF8StringEncoding]];
[savedFile synchronizeFile];
[savedFile closeFile];
```

Lesen

Um die gespeicherte Datei wieder einzulesen, wird wieder ein `NSFileHandle`-Objekt für die gespeicherte Datei erzeugt. Die Methode `readDataToEndOfFile` liest den Inhalt dann ein.

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSString *documentsPath = [NSSearchPathForDirectoriesInDomains
   (NSDocumentDirectory, NSUserDomainMask, YES) objectAtIndex:0];
NSString *filePath = [documentsPath stringByAppendingPathComponent:
   @"myFile.txt"];

NSFileHandle *fileHandle = [NSFileHandle
fileHandleForReadingAtPath:filePath];
NSData *data = [fileHandle readDataToEndOfFile];
NSString *string = [[NSString alloc] initWithData:data
   encoding:NSUTF8StringEncoding];
NSLog(string);
[string release];
```

Kopieren

Für das Kopieren mit der Methode `copyItemAtPath:toPath:error:` wird neben dem Pfad der zu kopierenden Datei natürlich noch der Pfad für die Kopie benötigt:

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSString *documentsPath = [NSSearchPathForDirectoriesInDomains
   (NSDocumentDirectory, NSUserDomainMask, YES) objectAtIndex:0];
NSString *fromPath = [documentsPath stringByAppendingPathComponent:
   @"myFile.txt"];
NSString *toPath = [documentsPath stringByAppendingPathComponent:
   @"newFile.txt"];

NSError *error = nil;
if (![fileManager copyItemAtPath:fromPath toPath:toPath error:&error]) {
   // Handle the error.
}
```

Löschen

Als letztes Beispiel für die Verwendung der Klasse `NSFileManager` soll das Löschen von Dateien mit der Methode `removeItemAtPath:error:` dienen.

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSString *documentsPath = [NSSearchPathForDirectoriesInDomains
   (NSDocumentDirectory, NSUserDomainMask, YES) objectAtIndex:0];
NSString *filePath = [documentsPath stringByAppendingPathComponent:
   @"myFile.txt"];

NSError *error = nil;
if (![fileManager removeItemAtPath:filePath error:&error]) {
   // Handle the error.
}
```

19.4 Property-Listen

Property-Listen sind Datenstrukturen, die aus Key/Value-Paaren und Listen von Werten aufgebaut sind. Beispielsweise fällt ein Array darunter, dessen Elemente aus *Dictionary*-Objekten bestehen. Aber auch andere Kombinationen und tiefere Verschachtelungen sind denkbar. Die folgenden Datentypen sind beim Erstellen der Liste zulässig:

- `NSArray`
- `NSDictionary`
- `NSData`
- `NSDate`
- `NSNumber`
- `NSString`

Hinzu kommen noch die entsprechenden `Mutable`-Klassen.

Strukturen dieser Art lassen sich natürlich auch in anderen Sprachen erstellen. Die Besonderheit in Cocoa liegt darin, dass solchermaßen aufgebaute Objekte von der API und der Entwicklungsumgebung eine großartige Unterstützung erfahren. Die komplexen Objekte lassen sich in das XML-Format hin- und wieder zurückverwandeln. Das XML kann gespeichert und bei Bedarf wieder geladen werden. Das folgende Listing zeigt eine Property-Liste im XML-Format.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Name</key>
    <string>John Doe</string>
    <key>Phones</key>
    <array>
        <string>408-974-0000</string>
        <string>503-333-5555</string>
    </array>
</dict>
</plist>
```

In Xcode können Property-Listen mit einem komfortablen Editor bearbeitet werden. Ein Beispiel für die Verwendung von Property-Listen sind *Settings*, die von der Klasse `NSUserDefaults` verwaltet werden. Mehr dazu im nächsten Kapitel, in dem auch der Editor benutzt wird.

In diesem Kapitel interessiert das Speichern und Laden von Property-Listen bzw. deren Bestandteilen. Die Klassen `NSArray`, `NSDictionary`, `NSData` und `NSString` (zuzüglich ihrer `Mutable`-Varianten) können leicht über ihre Methode `writeToFile:atomically:` gespeichert werden. Im folgenden Listing wird ein Array mit drei Strings erzeugt und anschließend mithilfe dieser Methode im *Documents*-Verzeichnis der App abgelegt:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
   NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
NSString *filename = [documentsDir
   stringByAppendingPathComponent:@"Array.txt"];

NSMutableArray *array = [[NSMutableArray alloc]init];
[array addObject:@"Rot"];
[array addObject:@"Grün"];
[array addObject:@"Blau"];

[array writeToFile:filename atomically:YES];
[array release];
```

Es lohnt sich, die Datei mit dem Finder in oben genanntem Verzeichnis aufzuspüren und anzuzeigen. Wie das folgende Listing zeigt, handelt es sich bei dem Inhalt der Datei in der Tat um das generische XML-Format für Property-Listen:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
   <string>Rot</string>
   <string>Grün</string>
   <string>Blau</string>
</array>
</plist>
```

Natürlich kann das Array genauso einfach wieder geladen werden. Die dafür zuständige Klassenmethode heißt `arrayWithContentsOfFile:`:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
   NSUserDomainMask, YES);
NSString *documentsDir = [paths objectAtIndex:0];
NSString *filename = [documentsDir
   stringByAppendingPathComponent:@"Array.txt"];

NSMutableArray *array = [NSMutableArray arrayWithContentsOfFile:filename];

for (NSString *string in array)
{
   NSLog(string);
}
```

20 Settings

Benutzerspezifische Einstellungen müssen oft dauerhaft gespeichert werden. Grundsätzlich stehen dafür zwei Realisierungsmöglichkeiten zur Auswahl.

Die Einstellungen könnten zum einen in der *Einstellungen*-App platziert werden. Hier, in den System-Settings, finden sich die Einstellungen verschiedenster Programme wie *Mail* oder *Safari* in trauter Runde:

Bild 20.2: Verschiedene Apps in den Settings

Alternativ dazu könnte auch ein eigener Settings-Bereich, etwa in Form eines modalen Views, angeboten werden. Da diese Settings aus der eigenen App aufgerufen werden können, spricht man hier von *In App Settings*.

Anwendungen vom Typ Utility (zum Beispiel *Aktien* oder *Wetter*) haben grundsätzlich eigene Einstellungen auf dem Flipside View (also der »Rückseite«) der App.

Sowohl System-Settings als auch In App Settings haben ihre Berechtigung und sind Inhalt dieses Kapitels.

20.1 System-Settings

System-Settings sollten dann favorisiert werden, wenn die Einstellungen nie oder zumindest selten verändert werden müssen. Ein Beispiel hierfür könnten Namen, Passwörter oder Mailadressen sein, die in der Anwendung gespeichert werden sollen.

Zum Erzeugen von Einträgen in die System-Settings muss kein Objective-C-Code geschrieben werden. Stattdessen wird ein *Settings Bundle* erstellt und angepasst. Die Einstellungen des Benutzers werden automatisch in das Bundle übertragen und können von dort auch wieder gelesen werden.

Ein neues Settings Bundle wird über *File > New File…* und die anschließende Auswahl von *Settings Bundle* (Reiter *Ressource*) im Dialog angelegt. Den Dateinamen belässt man am besten auf `Settings.bundle`.

Bild 20.1: Erstellen eines Settings Bundles

Das Settings Bundle ist eine Property-Liste und wird als solche in der *Groups & Files*-Ansicht von Xcode mit einem kleinen Dreieck neben dem Symbol dargestellt:

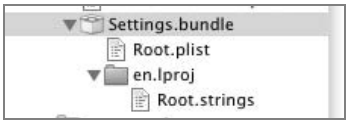


Bild 20.2: Settings Bundle in Xcode

Das Dreieck deutet darauf hin, dass es sich hier um einen Container handelt, der noch weiter aufgeklappt werden kann. Das Bundle enthält die Property-Liste `Root.plist` und einen Ordner `en.lproj` mit der Datei `Root.strings`. Letztere Datei beinhaltet die englischsprachigen Versionen von Texten in den Settings. Durch Klick auf `Root.plist` wird der Editor zum Eintragen der Settings geöffnet:

Key	Type	Value
▼Root	Dictionary	(2 items)
StringsTable	String	Root
▼PreferenceSpecifiers	Array	(4 items)
▶Item 1	Dictionary	(2 items)
▶Item 2	Dictionary	(8 items)
▶Item 3	Dictionary	(4 items)
▶Item 4	Dictionary	(7 items)

Bild 20.3: Settings im Property-Listen-Editor

Entscheidend sind die Einträge unter *PreferenceSpecifiers*. Bei den vier bereits vorhandenen Items handelt es sich um Beispieleinträge, deren Auswirkungen Sie bei Interesse sofort in der *Einstellungen*-App in Augenschein nehmen können. Ein neuer Eintrag mit dem Namen des Projekts ist hinzugekommen:

Bild 20.4: Neuer Eintrag in den Settings

Ein Tap darauf zeigt eine Tabelle im *Grouped*-Style, auf der drei Einstellungen via Textfeld, Switch und Slider getätigt werden können:

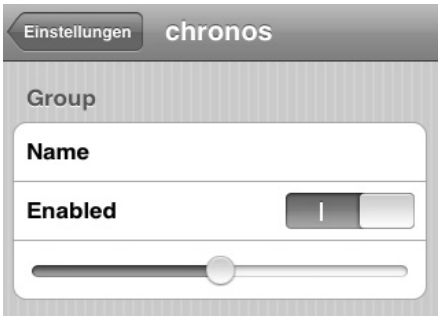
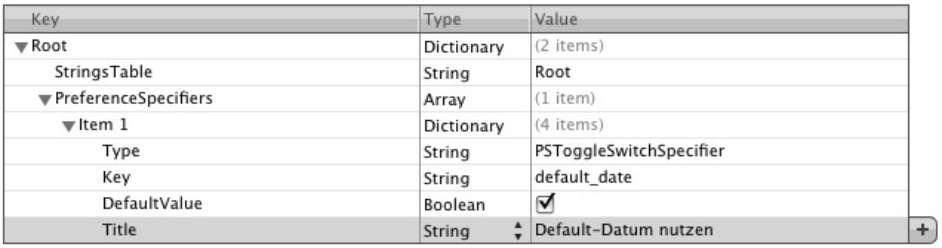


Bild 20.5: Beispiel-Settings

Das ist zwar schon sehr beeindruckend, wird aber in den seltensten Fällen den eigenen Anforderungen entsprechen. Um unsere eigenen Settings zu machen, löschen wir also zunächst die vier Items in `Root.plist`.

Anschließend wird mittels des kleinen Buttons am rechten Rand der Tabelle oder per Kontextmenü (*Add row*) ein neues Item zugefügt und der Typ auf *Dictionary* gesetzt. Das neue Item muss sich in der Hierarchie unterhalb von *PreferenceSpecifiers* befinden. Dem Dictionary können nun wiederum Items zugefügt werden. Dazu wird das Dreiecksymbol des Dictionary-Ordners *Item1* aufgeklappt. Das Pluszeichen am rechten Rand verwandelt sich in das Symbol zum Zufügen neuer Dictionary-Einträge.

Für unsere App benötigen wir einen Toggle-Button, der darüber entscheidet, ob das heutige Datum als Default-Datum beim Erfassen neuer Zeiten angenommen werden soll oder das Datum nicht gesetzt wird. Dazu werden die folgenden Items benötigt:



Key	Type	Value
Root	Dictionary	(2 items)
StringsTable	String	Root
PreferenceSpecifiers	Array	(1 item)
Item 1	Dictionary	(4 items)
Type	String	PSToggleSwitchSpecifier
Key	String	default_date
DefaultValue	Boolean	☑
Title	String	Default-Datum nutzen

Bild 20.6: Settings für das Default-Datum

Title ist der Text, der in den Settings angezeigt wird.

Über *Type* wird das anzuzeigende Control ausgewählt. Gültige Werte sind:

- `PSTextFieldSpecifier` (Text Field)
- `PSTitleValueSpecifier` (Label)
- `PSToggleSwitchSpecifier` (Switch)
- `PSSliderSpecifier` (Slider)
- `PSMultiValueSpecifier` (Auswahlliste)
- `PSGroupSpecifier` (zur Gruppierung von Einträgen)
- `PSChildPaneSpecifier` (Verweis auf separates List-File)

Der *Key* erlaubt den Zugriff auf die Einstellung aus dem Code heraus. Abhängig vom Typ des Controls können weitere Key/Value-Paare wie zum Beispiel *DefaultValue* angegeben werden.

Eine weitere Einstellmöglichkeit wird für die Mail-Adresse des Benutzers benötigt. Sie dient beispielsweise als Vorbelegung für das Versenden von CSV-Reports. Erforderlich sind hierfür die folgenden Einträge:

Key	Type	Value
Root	Dictionary	(2 items)
StringsTable	String	Root
PreferenceSpecifiers	Array	(2 items)
Item 1	String	
Item 2	Dictionary	(4 items)
Type	String	PSTextFieldSpecifier
Key	String	mail_prefernce
KeyboardType	String	EmailAddress
Title	String	Email-Adresse

Bild 20.7: Settings für die Mail-Adresse

Obige Einstellungen führen zu folgenden Einträgen in der Settings-Anwendung:

Bild 20.8: Neue Settings-Einträge

Die Einstellungen müssen nun natürlich irgendwie im Applikationscode abrufbar sein. Der Zugriff auf die Settings erfolgt über den *Key* mithilfe der Klasse `NSUserDefaults`. Sie ist die Schnittstelle zu den *User Default Settings* und verfügt über eine Reihe von Methoden, um die Einstellungen für einen bestimmten Key im gewünschten Datentyp zu erhalten. Für einen Switch erhält man den Wert als Boole'schen Wert über die Methode `boolForKey:`

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
BOOL value = [defaults boolForKey:@"default_date"];
```

Der Wert des E-Mail-Textfelds wird durch folgenden Code zurückgegeben:

```
NSString *mail = [defaults stringForKey:@"mail_preference"];
```

Wie in den Systemeinstellungen zu erkennen ist, können die Tabellenzellen für jede Anwendung ein kleines Icon (29 x 29) im PNG-Format enthalten. Dieses kann optional unter dem Namen `Icon-Settings.png` zur Verfügung gestellt werden. Wird es nicht gefunden, wird stattdessen das Haupt-Icon der Anwendung skaliert und verwendet.

20.2 In App Settings

In App Settings sind immer dann sinnvoll, wenn die Einstellungen häufig geändert werden müssen oder aber die Anwendung zum Ändern der Einstellungen nicht verlassen werden soll. Stellen Sie sich eine Umkreissuche, beispielsweise nach Restaurants, vor. Wenn im eingestellten Umkreis nichts gefunden wird, möchten Sie eventuell (abhängig von der Größe des Lochs in Ihrem Bauch) den Suchradius vergrößern. Dazu die Anwendung zu verlassen, in die Settings zu navigieren, die Einstellung zu tätigen und die Anwendung wieder zu starten ist – ohne Frage – keine sehr komfortable Benutzerführung. Hier scheint eine Einstellmöglichkeit im Programm gerechtfertigt.

20.2.1 In App Settings selbst entwickeln

Wenn man den *Settings*-View selber gestaltet, sollte man im Sinne eines einheitlichen Look & Feel das Aussehen an die System-Settings anlehnen. Das bedeutet, es sollten möglichst ein Table View im Style *Grouped* verwendet und die Bedienelemente im *Accessory*-View angezeigt werden. Unsere Zeiterfassung verwendet in der endgültigen

Version keine eigenen In App Settings, dennoch wird hier kurz am Beispiel des Default-Datums demonstriert, wie die Entwicklung ablaufen würde.

Ziel ist der folgende *Settings*-View:

Bild 20.9: Ein eigener Settings-Dialog

Zunächst muss der eigene *Settings*-View irgendwo aufgerufen werden können. Dazu bietet sich ein *Tool Bar*-Button auf dem Hauptmenü-Screen an. Das Hinzufügen des Buttons erfolgt in der Methode `viewDidLoad` der Klasse `MenuTableViewController`:

```
[self setToolbarItems:[NSArray arrayWithObjects:
   [[[UIBarButtonItem alloc] initWithImage:[UIImage
     imageNamed:@"settings_small.png"] style:UIBarButtonItemStylePlain
     target:self action:@selector(showSettings)]autorelease],
...
```

In der zugehörigen Action-Methode `showSettings` wird ein neu erzeugter Table View Controller als modaler View Controller, eingebettet in einen ebenfalls neu erzeugten Navigation Controller, angezeigt:

```
- (void)showSettings {
   UIViewController *controller = [[InAppSettingsViewController alloc]
     initWithNibName:@"SettingsView" bundle:nil];
   UINavigationController *navController = [[UINavigationController alloc]
     initWithRootViewController:controller];
   controller.navigationItem.leftBarButtonItem = [[[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemCancel
      target:self action:@selector(cancelInfo)] autorelease];
   controller.navigationItem.rightBarButtonItem = [[[UIBarButtonItem alloc]
     initWithBarButtonSystemItem:UIBarButtonSystemItemSave
       target:controller action:@selector(saveSettings)] autorelease];
   [self.navigationController presentModalViewController:navController
   animated:YES];
   [controller release];
   [navController release];
}
```

Der umgebende Navigation Controller wird benutzt, um in der zugehörigen Navigation Bar Buttons für *Abbrechen* und *Sichern* anzeigen zu können.

Bei der Klasse `InAppSettingsViewController` handelt es sich um einen Table View Controller. Er wird, wie schon mehrmals besprochen, zusammen mit dem Nib-File `InAppSettingsView` erzeugt.

In der Methode `tableView:cellForRowAtIndexPath:` wird die einzige Zelle für die Settings konfiguriert. Sie erhält das Label und als *Accessory*-View einen Switch. Die Stellung des Switch (*An* oder *Aus*) wird mithilfe eines Wertes aus einem Dictionary gesetzt:

```
...
switch (indexPath.row) {
   case 0: {
     cell.textLabel.text = @"Default-Datum";
     NSString *defaultDate =  (NSString *)[dict
objectForKey:@"default_date"];
     switcher.on = [defaultDate boolValue];
     cell.accessoryView = switcher;
   }
}
...
```

Das Dictionary wird vor dem Anzeigen des Views mithilfe der Methode `dictionaryWithContentsOfFile:` aus dem Dateisystem geladen. Die analoge Methode der Klasse `NSArray` wurde im letzten Kapitel vorgestellt. Als Speicherort dient hier der *Documents*-Ordner im Home-Verzeichnis der App. Wird beim Versuch, die Property-Liste zu laden, keine Datei gefunden, wird das Dictionary neu erzeugt:

```
- (void)viewWillAppear:(BOOL)animated {
   dict = [NSMutableDictionary dictionaryWithContentsOfFile:filename];
   if(dict == nil)
   {
     NSArray *keys = [NSArray arrayWithObjects:@"default_date", nil];
     NSArray *objects = [NSArray arrayWithObjects: @"NO", nil];
     dict = [NSMutableDictionary dictionaryWithObjects:objects
forKeys:keys];
   }
   [dict retain];
   [super viewWillAppear:animated];
}
```

Schließlich muss beim Betätigen des *Speichern*-Buttons die Stellung des Switchs in das Dictionary geschrieben und dieses abgespeichert werden:

```
- (void)saveSettings {
   NSString *switchValue = @"YES";
   if (switcher.on == NO) switchValue = @"NO";
   [dict setObject:switchValue forKey:@"default_date"];
   [dict writeToFile:filename atomically:YES];
   [self.navigationController dismissModalViewControllerAnimated:YES];
}
```

20.2.2 Fertige Frameworks

Das Anzeigen von In App Settings ist ein Standardproblem, das natürlich schon von einigen Entwicklern bearbeitet wurde. Deshalb existieren fertige Frameworks, die die Arbeit erleichtern können. Im Folgenden wird *InAppSettings* [URL-INAPPSETTINGS] vorgestellt. Eine Alternative dazu stellt das *InAppSettingsKit* [URL-INAPPSETTINGSKIT] dar.

Wie in den beiden vorhergehenden Abschnitten gesehen, muss der Entwickler sich entscheiden, ob er die Einstellungen als System-Setting in der Settings App anbietet oder als In App Setting direkt in der App vornehmen lässt. Muss er das wirklich? Eine dritte Variante ist denkbar. Die Settings könnten als System-Settings gespeichert, aber trotzdem zusätzlich in der App angeboten werden. Der View Controller muss dazu seine Daten aus dem Settings Bundle der App lesen. Genau diesen Weg beschreitet das Projekt *InAppSettings*.

Nach dem Download und dem Entpacken der Sourcen werden diese dem eigenen Projekt in Xcode zugefügt. Am einfachsten geht das durch Drag & Drop des *InAppSettings*-Ordners aus dem Finder direkt in die Xcode *Groups & Files*.

Die Verwendung des Frameworks ist erfreulicherweise sehr einfach. Nach dem Import von `InAppSettings.h` wird an geeigneter Stelle, etwa in einer Action-Methode, eine Instanz von `InAppSettingsModalViewController` erzeugt. Anschließend wird der View Controller zum Beispiel als modaler View angezeigt:

```
InAppSettingsModalViewController *settings =
   [[InAppSettingsModalViewController alloc] init];
[self presentModalViewController:settings animated:YES];
[settings release];
```

Das war`s schon. Die zuvor als Systemeinstellungen angelegten Settings sind in der App verfügbar.

Die folgende Abbildung zeigt einen Screenshot des View Controllers *InAppSettings* für die *Chronos*-Einstellungen:

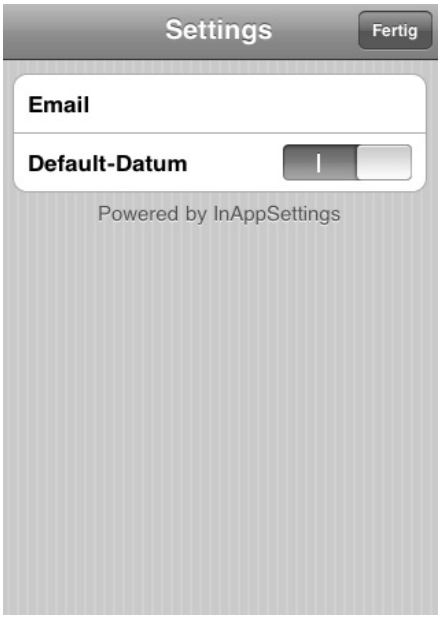


Bild 20.10: InAppSettings-Framework

Wie man sieht, versucht das Framework die System-Settings möglichst ähnlich nachzubilden.

Ob diese Art der Einstellungsverwaltung für Ihre App sinnvoll ist, hängt, wie schon erläutert, von der Art der Settings ab. Selten oder nie zu verändernde Einstellungen stören in der App eher und erschweren das Auffinden von oft benötigen Einstellungen. Für viele Anwendungen ist das aber ein guter Kompromiss. Weitere Informationen zur Benutzung des Frameworks finden sich auf der oben genannten Webseite.

21 Lokalisierung & Internationalisierung

Über den genialen Verteilungsmechanismus App Store ist es eine Leichtigkeit, Benutzer in aller Welt zu erreichen. Voraussetzung dafür ist natürlich, dass die Benutzer Ihre App auch verstehen. Die Anwendung muss dazu mindestens in die Zielsprachen übersetzt werden. In vielen Fällen sind weitere Anpassungen, beispielsweise an Währungs- oder Datumsformate, notwendig.

21.1 Welche Sprachen sind sinnvoll?

Um abschätzen zu können, ob eine Sprachvariante sinnvoll ist, hilft es sehr, die Verteilung der iPhone- und iPod-touch-Benutzer auf die verschiedenen Länder zu kennen. An diese Daten ist recht schwer heranzukommen. Eine ganz gute Quelle ist der Mobile Metrics Report [URL-METRICS] des Unternehmens *Admob*. Im Juni 2009 etwa sah die Verteilung folgendermaßen aus:

United States	54%
United Kingdom	7%
Germany	6%
France	5%
Canada	4%
Australia	3%
Japan	2%
Italy	2%
Mexico	1%
Spain	1%
Others	15%

Quelle: admob (*http://metrics.admob.com*)

Wie man an den Zahlen erkennen kann, ist der englischsprachige Raum (USA, UK, Kanada, Australien) mit über 60 Prozent Marktanteil auf keinen Fall zu vernachlässigen. Zusammen mit dem deutschsprachigen Raum erreicht man gut 70 Prozent der iPhone-OS-Nutzer. Für die Zeiterfassung beschränken wir uns deshalb neben der ohnehin vorhandenen deutschen Version auf eine englischsprachige Variante. Ob sich die (Übersetzungs-)Kosten und Mühen für eine Anpassung an weitere Sprachen lohnen, muss jeder Entwickler selber abschätzen.

21.2 Internationalisierung

Die Internationalisierung, oft auch mit *i18n* (die 18 steht für die Anzahl der Buchstaben zwischen i und n im englischen Wort »Internationalization«) abgekürzt, befasst sich mit der Übersetzung der Texte in der Anwendung.

Kennen Sie das berühmte Zitat »Equal goes it loose«? Auch wenn der deutsche Bundespräsident Lübke die ihm zugeschriebene unsägliche Übersetzung (gemeint war »Gleich geht's los«) nicht wirklich benutzt hat, macht es doch eines klar: Texte können nicht wörtlich übersetzt werden. Tun Sie also all Ihren Benutzern einen Gefallen und lassen Sie Ihre Übersetzungen nicht von Tools im Internet erstellen.

Texte finden sich nicht nur in Form von Strings in Programmcode, sondern beispielsweise auch in Grafiken oder Nib-Files. Die verschiedenen Versionen der Dateien werden in einem separaten Verzeichnis für jede Sprache aufbewahrt. Dazu werden zunächst die zwei Verzeichnisse *de.lproj* und *en.lproj* innerhalb des Projektordners *trunk* angelegt.

21.2.1 Strings

Bisher wurden alle Strings für die Anwendung direkt im Programmcode angegeben. Um mehrsprachige Varianten zu erzeugen, werden die Strings aus dem Code in separate Dateien ausgelagert. Die Files mit Namen `Localizable.strings` werden in Xcode mit *File > New File... > Other > Strings File* angelegt:

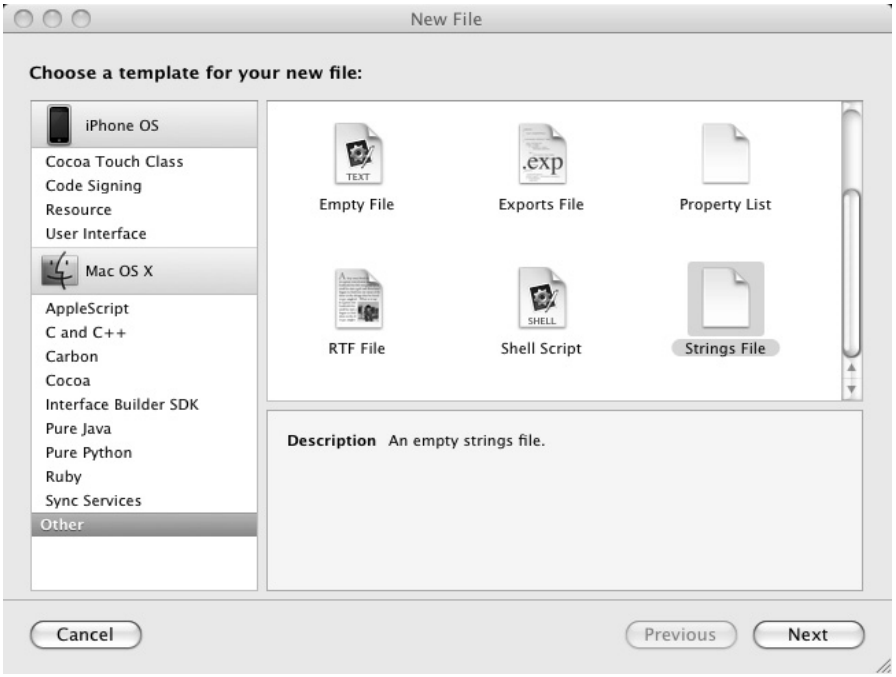


Bild 21.1: Erzeugen von `Localizable.strings`

Als Zielordner sind *en.lproj* und *de.lproj* auszuwählen (jeder Ordner erhält ein *Strings File*). Nach dem Anlegen der beiden Dateien in den beiden Ordnern sollte Xcode unter *Resources* einen Unterordner *Localizable.strings* mit den Einträgen *de* und *en* enthalten:

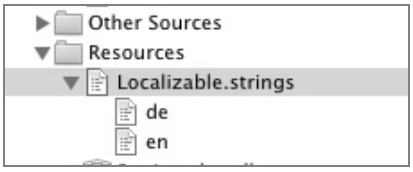


Bild 21.2: Ordnerstruktur für die Übersetzung

Die Dateien *de* und *en* können nun in Xcode wie gewohnt markiert und bearbeitet werden. Sie enthalten die Übersetzungen in Form von Key/Value-Paaren. Key und Value müssen in doppelten Anführungsstrichen stehen und werden durch ein Gleichheitszeichen getrennt. Die Zeile wird mit einem Semikolon abgeschlossen. Die Datei *de* erhält beispielsweise folgenden Eintrag:

```
"KundenKey" = "Kunden";
```

In der Datei *en* findet sich dagegen Folgendes:

```
"KundenKey" = "Customers";
```

Der Zusatz *Key* ist übrigens nicht erforderlich. Allerdings ist er ganz hilfreich, um im Code die Keys von anderen Strings unterscheiden zu können und zu sehen, wo die

Übersetzung noch nicht funktioniert. Immer wenn Sie in der laufenden App einen Text finden, der auf *Key* endet, liegt noch ein Problem vor.

Um nun im Anwendungscode auf die richtige Variante des Keys zuzugreifen, wird die Funktion `NSLocalizedString` verwendet. Sie erhält als ersten Parameter den Key und als zweiten Parameter optional einen Kommentar:

```
NSString *NSLocalizedString(NSString *key, NSString *comment)
```

Überall im Code, wo bisher ein Text als String vergeben wurde, muss der Funktionsaufruf mit dem passenden Key eingefügt werden.

Für das Hauptmenü sieht der entsprechende Abschnitt der Methode `tableView: cellForRowAtIndexPath:` damit folgendermaßen aus:

```
...
   cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;
   switch (indexPath.row) {
        case 0: {
            cell.textLabel.text = NSLocalizedString(@"KundenKey", nil);
        cell.imageView.image = [UIImage imageNamed:@"users_small.png"];
        } break;
        case 1: {
     cell.textLabel.text = NSLocalizedString(@"ProjekteKey", nil);
     cell.imageView.image = [UIImage imageNamed:@"clipboard_small.png"];
        } break;
...
```

Zum Testen der Internationalisierung stellt man die Sprache des Geräts um. Die Möglichkeit dazu hat man auf dem iPhone oder im Simulator unter *Einstellungen* > *Allgemein* > *Landeseinstellungen* > *Sprache*. An dieser Stelle kann außerdem die Region verändert werden. Wir werden später noch davon Gebrauch machen. Beachten Sie bitte auch das Beispiel für das regionale Format, das unter der *Region*-Zelle angezeigt wird:

Bild 21.3: Umstellung von Sprache und Region

Sollten Sie unsicher sein, wie man etwa in Korea Datum, Uhrzeit oder Telefonnummer formatiert, können Sie hier nachsehen.

Nach der Änderung der Sprache auf Englisch präsentiert sich das Hauptmenü folgendermaßen:

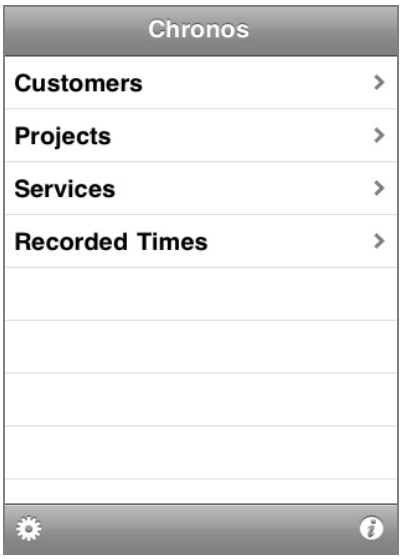


Bild 21.4: Hauptmenü nach i18n

Wenn Sie ein wenig weiter in der Anwendung umhernavigieren, werden Sie feststellen, dass sich nach der Sprachumstellung auch einige Texte geändert haben, die nicht in den Key/Value-Paaren enthalten sind. Dies betrifft Texte, die nicht vom Entwickler vergeben wurden, sondern in den Framework-Klassen enthalten sind. Der Edit-Button trägt zum Beispiel je nach Spracheinstellung die Aufschrift *Edit* oder *Bearbeiten*.

Ebenfalls ein String, wenn auch ein ganz besonderer, ist der Titel der App, der unter dem Icon auf dem Home-Screen angezeigt wird. Um ihn abhängig von der Sprache zu definieren, wird ein zweites File in den Sprachordnern *de.lproj* und *en.lproj* benötigt. Es wird ebenfalls über den *New File*-Assistenten angelegt, ist auch vom Typ *Strings File* und bekommt den Namen `InfoPlist.strings`. Wie `Localizable.strings` muss auch dieses File in beiden Sprachordnern existieren. In der Datei kann über den Key `CFBundleDisplayName` der Name der App in der jeweiligen Sprache angegeben werden:

```
CFBundleDisplayName = "Chronos";
```

21.2.2 Grafiken

Auch Grafiken können Texte beinhalten. Ist das der Fall, dann müssen auch sie in mehreren Sprachversionen vorgehalten werden. Im Prinzip funktioniert das genau wie bei den Strings. Die Grafik wird mit gleichem Namen einmal in einen Ordner *en.lproj* und einmal in den Ordner *de.lproj* im Dateisystem gepackt. Anschließend werden die

beiden Dateien in Xcode über die Option *Add > Existing Files* aus dem Kontextmenü von *Groups & Files* zugefügt. Xcode bemerkt dabei, dass es sich um das gleiche File in verschiedenen Sprachvarianten handelt, und visualisiert die Datei wie folgt:

Bild 21.5: Mehrsprachige Grafik

Beim Benutzen der Grafik im Code werden die Unterordner nicht berücksichtigt:

```
UIImage *image = [UIImage imageNamed:@"star.png"];
imageView.image = image;
```

Abhängig von der eingestellten Sprache wird automatisch die passende Grafik ausgewählt.

21.2.3 Nib-Files

Schließlich können Nib-Files Texte enthalten. Das kann zum Beispiel der Titel eines Views oder die Beschriftung eines Labels sein. Für die Übersetzung des Nib-Files muss wie bei den zuvor besprochenen Texten im Code oder in den Grafiken eine Kopie der Datei in einem Ordner mit Namen des Sprachkürzels angelegt werden. Die kopierten Nib-Files werden dann wie gewohnt mit dem Interface Builder geöffnet und an den entsprechenden Stellen die Texte in der gewünschten Sprache eingetragen.

Sie finden, dass es keine gute Idee ist, bei zehn verschiedenen Sprachen zehn Kopien des Nib-Files vorzuhalten? Damit liegen Sie vollkommen richtig. Im Gegensatz zu Grafiken oder Texten handelt es sich um komplexe GUI-Beschreibungen, die mit Sicherheit irgendwann angepasst werden müssen. Alle Änderungen müssen dann an zehn Dateien durchgeführt (und auch zehnmal getestet) werden. Falls separate Nib-Files für das iPad benutzt werden, darf das Ganze noch verdoppelt werden. Zudem sind die Texte verstreut im ganzen Nib-File und können nicht so einfach an einen Übersetzer geschickt werden, wie das mit einer einfachen Key/Value-Liste machbar ist. Insgesamt klingt das nicht nach einer praktikablen Lösung.

Wie damit umgehen? Wenn von vornherein klar ist, dass die App mehrsprachig sein wird, versucht man am besten, keine Texte in den Nib-Files zu vergeben. Die Texte können auch im Programmcode gesetzt werden. Dazu müssen natürlich alle zu beschriftenden Controls als Outlets auch im Code vorliegen. Letztlich macht sich dieser Aufwand aber bezahlt. Nachteil dieser Variante ist, dass man keinen Einfluss auf die Größe der Controls hat. Deutsche und französische Beschriftungen sind grundsätzlich länger als englische Beschriftungen, weil sie mehr Buchstaben für das gleiche Wort benötigen. Wenn diese kleine Unsauberkeit sehr stört, kann man versuchen, die Größe von Controls abhängig von der gewählten Sprache zu verändern – oder eben doch sprachabhängige Nib-Files verwenden.

Letztlich bleibt das Thema eines der wenigen Ärgernisse in der ansonsten so gut durchdachten iPhone-Programmierung.

21.2.4 Settings

Wie im vorigen Kapitel besprochen, werden System-Settings in einem *Settings Bundle* gespeichert. Die folgende Abbildung zeigt zur Erinnerung noch mal den Aufbau des Settings Bundles:

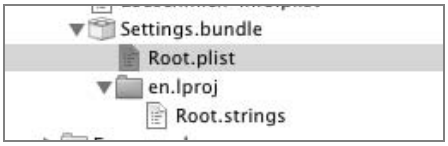


Bild 21.6: Settings Bundle

Das Bundle ist für eine Übersetzung schon vorbereitet. Die in `Root.plist` vergebenen Titel lassen sich in der Datei `Root.strings` in die gewünschte Sprache übersetzen. Um eine deutschsprachige Version der Datei anzulegen, spüren Sie die Datei zunächst durch Rechtsklick und Auswahl von *Reveal in Finder* auf und legen dann eine Kopie in dem Ordner *de.lproj* an. Beachten Sie bitte auch die Option *Paketinhalt anzeigen* beim Rechtsklick der Datei `Settings.bundle` im Finder. In Xcode sollte das Settings Bundle nach dem zur Aktualisierung erforderlichen Zu- und Aufklappen wie folgt aussehen:

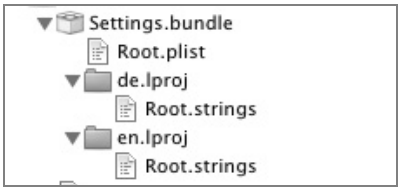


Bild 21.7: Settings Bundle mit neuem Ordner

Die *Title*-Property, die in `Root.plist` auf Default-Datum gesetzt wurde, kann nun in der englischsprachigen Variante der Datei `Root.strings` übersetzt werden:

```
"Default-Datum"="Default date";
```

Wie die folgende Abbildung zeigt, wird daraufhin bei ausgewählter englischer Sprache die englische Bezeichnung verwendet:

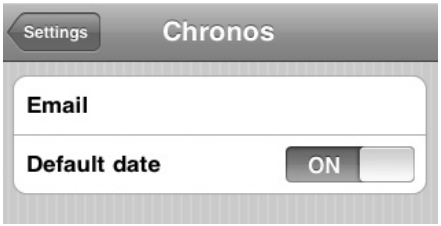


Bild 21.8: Übersetzte Settings

21.3 Lokalisierung

Die Internationalisierung ist relativ leicht zu realisieren, stellt aber leider nur eine Seite der Medaille dar. Die andere Seite ist die Lokalisierung. Man versteht darunter die Formatierung von Datum, Uhrzeit, Währung, Einheiten und Telefonnummern abhängig von der Region. Betrachten wir dazu ein kleines Beispiel.

In Deutschland sieht ein Datum mit Uhrzeit folgendermaßen aus:

Montag, 5. Januar 2009, 00:34

In den USA würde man aber stattdessen schreiben:

Monday, January 5, 2009, 12:34 AM

Der Monat steht hier vor dem Tag und statt der 24-Stunden-Schreibweise wird AM (ante meridiem) oder PM (post meridiem) verwendet.

Die Unterschiede betreffen aber noch weitere Bereiche. Die Währungsangabe $ steht in den USA vor dem Betrag, während sie in Deutschland üblicherweise dahintersteht. Der Dezimaltrenner ist in Deutschland ein Komma, Tausenderstellen werden durch einen Punkt getrennt. In den USA ist die Verwendung genau andersherum. Es gibt eine Unmenge solcher Unterschiede, die einem erst richtig bewusst werden, wenn man sich mit dem Thema beschäftigt.

21.3.1 Locale

Um all diesen Feinheiten gerecht zu werden, muss die in den Landeseinstellungen definierte Region berücksichtigt werden. Die Einstellungen werden in einem Objekt vom Typ `NSLocale` gespeichert. Eine *Locale* fasst eine Benutzergruppe zusammen, die die gleichen Erwartungen an die Sprachdarstellung hat. Der im *Locale*-Objekt enthaltene String hat die Form `LanguageCode_CountryCode`, also etwa `de_AT` oder `de_DE`. Der *Language Code* definiert die in der Region gesprochene Sprache, der *Country Code* erlaubt die Berücksichtigung von länderspezifischen Unterschieden trotz gleicher Sprache. Verwechseln Sie bitte nicht die von Ihnen eingestellte Sprache mit dem *Language Code*. Die gewählte Sprache ist die Sprache, in der Ihr iPhone mit Ihnen kommuniziert. Der *Language Code* dagegen ist die Sprache, die in einem Land gesprochen wird.

Man erhält die Locale durch Aufruf der Klassenmethode `currentLocale`:

```
NSLocale *currentUsersLocale = [NSLocale currentLocale];
NSLog(@"Current Locale: %@", [currentUsersLocale localeIdentifier]);
```

Für Deutschland ergibt das folgende Ausgabe:

```
Current Locale: de_DE
```

Die so erhaltene aktuelle Locale kann für die Formatierung regionabhängiger Daten verwendet werden. Es wäre natürlich sehr viel Arbeit, wenn man nun eigene Klassen für

die Formatierung schreiben müsste. Glücklicherweise sind diese für die meisten Anwendungszwecke schon fertig. Sie werden im weiteren Verlauf besprochen.

21.3.2 Zahlen: Number Formatter

Um Zahlen abhängig von den regionalen Gegebenheiten zu formatieren, verwendet man die Klasse `NSNumberFormatter`.

Beispiel:

```
NSNumber *number = [NSNumber numberWithFloat:40500.6];
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setNumberStyle:NSNumberFormatterDecimalStyle];
[numberFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
NSLog([numberFormatter stringFromNumber:number]);
```

Im Beispiel wird zunächst ein `NSNumber`-Objekt und anschließend ein `NSNumberFormatter` erzeugt. Der *Formatter* bekommt mitgeteilt, dass die Zahl im Dezimalstil formatiert werden soll.

Die Ausgabe lautet für die Region Deutschland: 40.500,602

Und für die Region Vereinigte Staaten: 40,500.602

Beachten Sie die Vertauschung von Punkt und Komma.

Um die gleiche Zahl als Währung zu formatieren, wird genauso vorgegangen, aber `NSNumberFormatterCurrencyStyle` als Stil gesetzt:

```
NSNumberFormatter *currencyFormatter = [[NSNumberFormatter alloc] init];
[currencyFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[currencyFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
NSLog([currencyFormatter stringFromNumber:number]);
```

Die Ausgabe lautet für die Region Deutschland: 40.500,60 €

Und für die Region Vereinigte Staaten: $40,500.60

Hier wird nicht nur das passende Währungssymbol verwendet, sondern auch die Position des Symbols wird angepasst.

Neben den beiden besprochenen existiert eine Reihe weiterer Stile:

- `NSNumberFormatterNoStyle`
- `NSNumberFormatterPercentStyle`
- `NSNumberFormatterScientificStyle`
- `NSNumberFormatterSpellOutStyle`

21.3.3 Datum: Date Formatter

Wie die Klasse `NSNumberFormatter`, so berücksichtigt auch `NSDateFormatter` die *Locale* automatisch. Damit lässt sich ein Datum, bei Bedarf mit Zeitangabe, leicht im regionalen Format darstellen. Für die Zeiterfassung wird genau das benötigt, etwa im *Detail*-View für die Zeiteinträge.

Der *Date Formatter* wird (in der Methode `viewDidLoad` der Klasse `ZeitenDetailTableViewController`) mit `alloc/init` erzeugt und danach mithilfe der `dateStyle`- und `timeStyle`-Properties konfiguriert:

```
dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateStyle:NSDateFormatterMediumStyle];
[dateFormatter setTimeStyle:NSDateFormatterNoStyle];
```

Die Verwendung erfolgt in der Methode `tableView:cellForRowAtIndexPath:` der gleichen Klasse:

```
...
   switch (indexPath.row) {
     case 0:
        cell.textLabel.text = NSLocalizedString(@"DatumKey", nil);
        cell.detailTextLabel.text = [dateFormatter
stringFromDate:zeit.datum];
       break;
...
```

Die Methode `stringFromDate:` des Date Formatters bekommt als Parameter das im `Zeit`-Objekt gespeicherte Datum übergeben.

Die folgende Abbildung zeigt das Ergebnis für die Regionen Deutschland und USA (und die Sprachen Deutsch und Englisch):

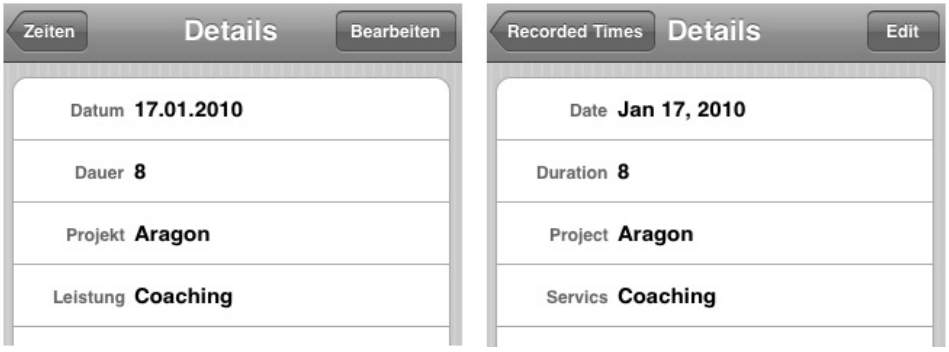


Bild 21.9: Date Formatter in der Zeiterfassung

21.3.4 Eigene Formatter

Die gängigen Anforderungen können mit den beiden Klassen sehr gut gelöst werden. Dennoch kann es je nach Anwendungsfall Datentypen geben, für die kein Formatter existiert. Ein Beispiel ist die Telefonnummer. In diesen Fällen hilft nur noch: selber entwickeln.

Um einen eigenen Formatter zu schreiben, wird eine Subklasse der abstrakten Klasse `NSFormatter` erzeugt. Bei der Initialisierung der Klasse wird die *Locale* ermittelt und in einer Instanzvariablen abgespeichert. In einer `stringFromXXX`-Methode wird sie dann verwendet, um eine der Region gerechte Repräsentation des übergebenen Datenwerts zurückzugeben.

22 Icons, Farben & Schriften

Die Zeiterfassung ist weitgehend fertig gestellt und kann bald den Weg in den App Store antreten. Es wird somit Zeit, ein wenig an der Optik zu feilen. Zugegebenermaßen ist das nicht unbedingt eine Aufgabe für Entwickler. Wir beschränken uns deshalb bei den im Folgenden beschriebenen »Aufhübschungen« auf ein paar grundlegende, einfache Dinge. Für größere kommerzielle Projekte ist es auf jeden Fall sinnvoll, einen Grafiker zurate zu ziehen. Gerade bei einem Gerät, das durch allerhand »Eye-Candy« wie Animationen und rotierbares Display begeistert, sind die Anwender durchaus designbewusst. Interessanterweise wirken die ersten iPhone-Programme von Apple wie *Mail* oder *Einstellungen* im Vergleich zu mancher neuen schicken App ein wenig in die Jahre gekommen. Ob's da wohl mal ein Facelifting geben wird?

22.1 Home-Screen-Icon

Das wichtigste Icon in der Anwendung ist das *Start*-Icon. Für den Benutzer immerzu präsent, harrt es auf dem Home-Screen geduldig aus, bis sich der Anwender endlich wieder mal zum Starten der zugehörigen App entschließt. Besonders wichtig ist das Icon aber nicht nur, weil es auf dem Home-Screen ständig sichtbar ist, sondern weil es auch im App Store zur Vermarktung der Anwendung genutzt wird. Kurzum, das Icon sollte gut aussehen.

Aus dieser einfachen Anforderung ergeben sich eine gute und eine schlechte Nachricht. Die gute ist: Wenn Sie nicht gerade ein talentierter Grafiker sind, fällt das Erstellen des Start-Icons nicht in Ihren Aufgabenbereich. Die schlechte ist, dass Sie eventuell jemanden für das Erstellen des Icons bezahlen müssen.

Eine interessante Alternative zum klassischen Designer sind Unternehmen wie *99designs* [URL-99DESIGNS] oder *designenlassen.de* [URL-DESIGNENLASSEN]. Das Geschäftsmodell dieser Unternehmen ist eine Art Wettbewerb mehrerer Grafiker für ein ausgeschriebenes Projekt. Der Auftraggeber entscheidet sich zwischen all den eingegangenen Vorschlägen für ein Design und erwirbt dieses somit. Der betreffende Designer erhält die ausgeschriebene Vergütung. Der Vorteil gegenüber dem herkömmlichen Ansatz, einen einzelnen Grafiker mit der Erstellung zu beauftragen, ist die große Anzahl von Vorschlägen (je nach ausgelobtem Preisgeld oft über 100), die zur Auswahl stehen.

Bei *99designs* ist ein einfaches Icon ab $149 zu haben. Dafür erhält man ca. acht Vorschläge.

Vom wem auch immer es erstellt wird – das Logo muss auf jeden Fall vom Typ *.png* sein und eine Größe von 57 x 57 Pixeln haben. Um das Abrunden der Ecken und den

sichelförmigen Schatteneffekt muss sich der Grafiker nicht kümmern, das passiert automatisch.

Das Icon wird dem Projekt unter *Resources* zugefügt. Der Name der Datei ist unter dem Key *Icon file* in `Info.plist` einzutragen:

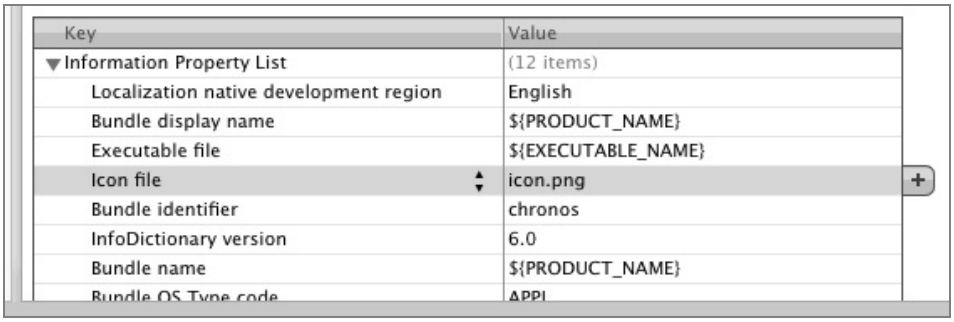


Bild 22.1: Eintrag des Icons in `Info.plist`

22.2 Icons & Grafiken in der App

Eine der einfachsten Möglichkeiten, eine Anwendung interessanter zu gestalten, sind Bilder. Getreu dem etwas angestaubten Motto »Ein Bild sagt mehr als tausend Worte« sollen auf jeden Fall die Menüpunkte des *Chronos*-Hauptmenüs durch Visualisierungen der Texte unterstützt werden. Die dafür verwendeten Icons müssen natürlich nicht nur gut aussehen, sondern sollten auch einen Bezug zum jeweiligen Menüpunkt haben.

Die für die Zeiterfassung verwendeten Icons entstammen einer kostenpflichtigen, jedoch relativ günstigen Icon-Sammlung, die speziell für das iPhone konzipiert wurde [URL-UIICONSET]. Im Internet findet man auch frei verfügbare Sammlungen. Vom Bilderklau, etwa über die Google-Bildersuche, ist wegen rechtlicher Konsequenzen dringend abzuraten.

Das Setzen der Icons erfolgt in der Methode `tableView:cellForRowAtIndexPath:` der Klasse `MenuTableViewController`:

```
...
switch (indexPath.row) {
   case 0: {
     cell.textLabel.text = @"Kunden";
     cell.imageView.image = [UIImage imageNamed:@"users_small.png"];
   } break;
   case 1: {
...
```

Vorher müssen die Icons natürlich dem Projektordner hinzugefügt worden sein.

Wie die folgende Abbildung zeigt, lässt dieser erste kleine Schritt das Menü schon deutlich interessanter erscheinen:

Bild 22.2: Menü mit Icons

Eine weitere sehr beliebte und von vielen Apps realisierte Verwendung von Grafik ist ein Logo in der Navigation Bar. Dem Namen der Anwendung kann dadurch mehr Gewicht verliehen werden. Wie im Kapitel »Navigation Controller« erklärt wurde, ist es möglich, den *Title View* einfach durch einen anderen View, zum Beispiel einen *Image View*, zu ersetzen. Das folgende Codestück zeigt, wie es geht. Es wird beispielsweise in der Methode `viewDidLoad` der betreffenden Klasse eingefügt:

```
UIImage *image = [UIImage imageNamed:@"logo.png"];
UIImageView *imageView = [[UIImageView alloc] initWithImage:image];
self.navigationItem.titleView = imageView;
[imageView release];
```

Ein gelungenes Beispiel für die Anpassung des Title View zeigt der folgende Screenshot der App *eCards*:

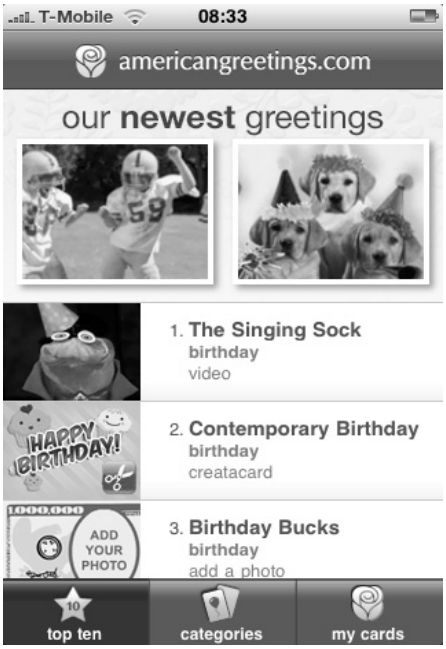


Bild 22.3: Angepasster `Title View`

22.3 Farben

Wie eingangs erwähnt, wirkt auf manchen Benutzer die blaugraue Default-Farbkombination ein wenig schmucklos. Eine einfach zu realisierende Alternative sind die »Farben« *black* und *translucent black*, die im Interface Builder für Statusbar, Navigation Bar und Tool Bar ausgewählt werden können. Zumindest *black* ist schon schicker, wenn auch recht düster:

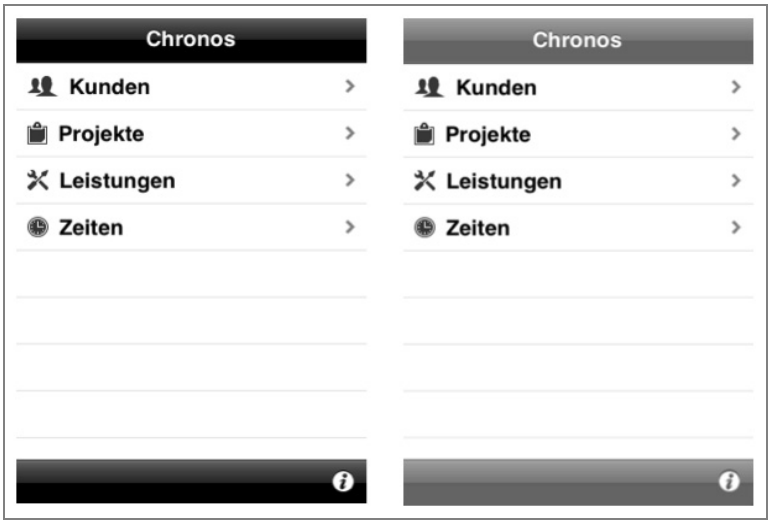


Bild 22.4: Chronos in Black und Translucent Black

Ein Beispiel für die Verwendung des Black-Farbschemas ist die *Xing*-Anwendung aus dem App Store.

Aufwendiger wird die Aufgabe, wenn die gesamte App nach eigenen Wünschen eingefärbt werden soll. Ob das überhaupt nötig ist, muss jeder für sich selbst entscheiden. Die bunten Anwendungen sehen im Vergleich zur graublauen Masse leicht fremdartig aus. Wenn die Farbwahl ansprechend ist, wird das vom Benutzer aber eher wohlwollend zur Kenntnis genommen. Der Schuss kann natürlich auch kräftig nach hinten losgehen. Auch dafür gibt es genügend Beispiele im App Store.

Wenn Sie nicht »Geschmack studiert« und auch sonst keine Ahnung von Farben haben, gibt es dennoch Möglichkeiten, zu einer ansprechenden Farbauswahl zu gelangen.

Eine Möglichkeit ist das Anlehnen an Farbschemata bestehender Anwendungen. Achten Sie bei allen Programmen, die Sie benutzen, auf die Farbwahl. Welche Farben wirken an welchen Stellen besonders gut? Wo muss eher eine dunkle und wo eher eine helle Farbe hin? Natürlich sollten Sie nicht gerade die Farbwahl Ihres härtesten Konkurrenten eins zu eins übernehmen. Bei über 150.000 Apps im App Store dürfte es aber keinen stören, wenn eine Anwendung, eventuell aus einem ganz anderen Bereich, ähnliche Farben verwendet.

Eine Alternative zum Orientieren an bestehenden Anwendungen bieten Webseiten, die geschmackvolle Farbpaletten vorstellen. Ein Beispiel dafür ist die Seite *Kuler* von Adobe [URL-ADOBE]. Die grafikinteressierte Community kann hier Farbvorschläge (Themes) zusammenstellen und zur Bewertung freigeben. Die Themes sind in Kategorien wie *Most popular* und *Highest rated* durchsuchbar.

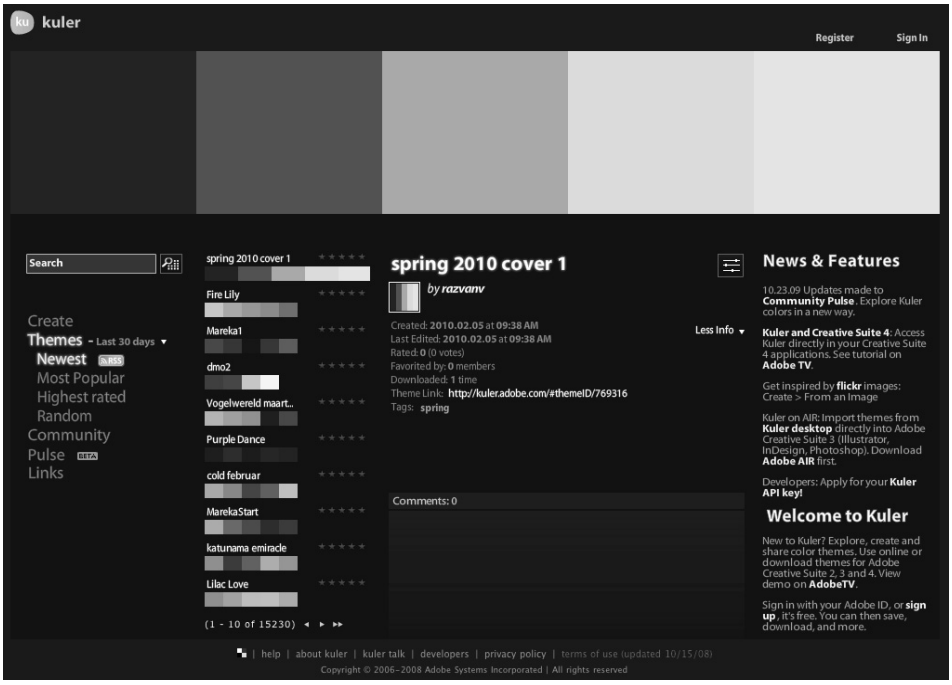


Bild 22.5: Kuler von Adobe

Hier sollte sich auf jeden Fall genügend Inspiration für ein eigenes Farbschema – oder vielleicht doch der Kontakt zu einem Designer – finden lassen. Mit einer Palette von vier zusammenpassenden, unterschiedlich hellen Farben (zuzüglich Schwarz und Weiß) lässt sich schon einiges machen.

Sind die Farben gefunden, geht es ans Kolorieren. Im Prinzip ist alles färbbar. Wir konzentrieren uns für das Buchprojekt auf einige wenige Elemente, deren Farbänderung besonders ins Auge sticht.

Zunächst sollen Navigation Bar und Tool Bar im gleichen Rot gefärbt werden. Dazu wird das entsprechende Nib-File (`MainWindow.xib`) im Interface Builder geöffnet und die Navigation Bar bzw. Tool Bar markiert. Die Farbe lässt sich durch einen Klick auf *Tint* im *Attributes*-Inspektor ändern:

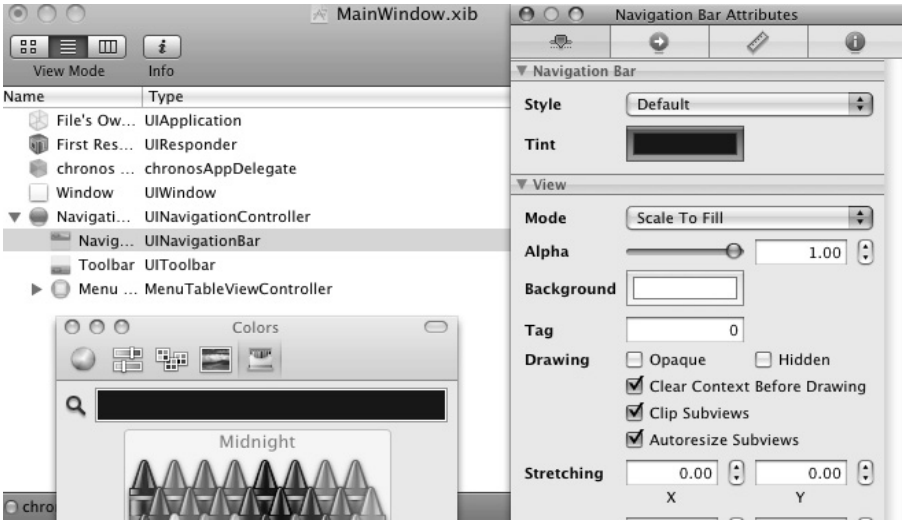


Bild 22.6: Ändern der Farbe im Interface Builder

Alternativ kann die Farbe auch im Code geändert werden. Zum Definieren von Farben wird die Klasse `UIColor` bzw. ihre statische Methode `colorWithRed:green:blue:alpha:` verwendet. Die Methode erwartet Werte von 0.0 bis 1.0. Falls die RGB-Werte, die Sie verwenden möchten, von 0 bis 255 reichen, müssen sie also noch durch 255 geteilt werden. Oft werden Farben auch in hexadezimaler Form angegeben. Die sechs Hex-Ziffern entsprechen drei RGB-Pärchen, die ebenfalls den maximalen Wert FF, also 255, annehmen können.

Beispiel:

Verwendet werden soll ein dunkelblauer Farbton mit einem Hexadezimal-Farbwert von #191726. Die Aufteilung in drei Pärchen ergibt 19 (25), 17 (23) und 26 (38). Teilt man die dezimalen Werte in den Klammern durch 255, ergeben sich die RGB-Werte zu 0.10, 0.09 und 0.15. Diese werden als Parameter der Methode `colorWithRed:green:blue:alpha:` verwendet:

```
UIColor *color = [UIColor colorWithRed:0.10 green:0.09 blue:0.15 alpha:1.0];
```

Die so definierten Farben müssen nun den entsprechenden UI-Komponenten zugewiesen werden.

Für die Navigation Bar heißt die entsprechende Property `tintColor`:

```
self.navigationController.navigationBar.tintColor =  color;
```

Analog wird mit der Tool Bar verfahren:

```
self.navigationController.toolbar.tintColor = [UIColor colorWithRed:0.10
green:0.09 blue:0.15 alpha:1.0];
```

Nachdem Navigation Bar und Tool Bar im neuen Farbton erscheinen, passt die blaue Farbe für die selektierte Zelle nicht mehr zu den roten Leisten. Hierfür soll nun ein

Orangeton verwendet werden. Um die Hintergrundfarbe einer selektierten Zelle zu ändern, muss der in `tableView:cellForRowAtIndexPath:` definierten Zelle ein neuer View, der `selectedBackgroundView`, zugewiesen werden. Der View wird nach dem Erzeugen mit der orangen Hintergrundfarbe versehen:

```
...
   if (cell == nil) {
        cell = [[[UITableViewCell alloc]
initWithStyle:UITableViewCellStyleDefault
     reuseIdentifier:CellIdentifier] autorelease];

   UIView *view = [[[UIView alloc] init] autorelease];
   view.backgroundColor = [UIColor colorWithRed:0.97 green:0.61
            blue:0.28 alpha:1.0];
   cell.selectedBackgroundView = view;
   }
...
```

Eine weitere Eigenschaft, die in Table Views leicht geändert werden kann, ist der Trennstrich zwischen den Zellen, der *Separator*. Die entsprechende Property heißt `separatorColor`:

```
((UITableView *)self.view).separatorColor = [UIColor colorWithRed:0.10
green:0.09 blue:0.15 alpha:1.0];
```

Das Hauptmenü und die *Listen*-Views sehen nach diesen Änderungen schon ganz brauchbar aus. Als Nächstes kommen die *Detail*-Views an die Reihe. Hier wird die Hintergrundfarbe der Tabellen (Property `backgroundColor`) in einen beigen Farbton geändert. Die neue Farbe erscheint rings um die Tabelle, die wegen Verwendung des *Grouped*-Style dort über einen schmalen Rand verfügt.

```
self.tableView.backgroundColor = [UIColor colorWithRed:1 green:0.90
blue:0.64 alpha:1.0];
```

In gleicher Weise wird mit dem Tabellenhintergrund des *Bearbeiten*-Views verfahren.

Die Farbspiele sind damit fast beendet. Das letzte störende Element sind die blauen Textlabels in den *Detail*-Views. Hier wird die Property `textColor` der Labels zur Farbänderung verwendet:

```
cell.textLabel.textColor = [UIColor colorWithRed:0.125 green:0.125
blue:0.149 alpha:1.0];
cell.detailTextLabel.textColor = [UIColor colorWithRed:0.125 green:0.125
blue:0.149 alpha:1.0];
```

Die folgende Tabelle führt zur besseren Übersicht noch mal alle geänderten Properties in den entsprechenden Klassen auf:

Zu ändernde Eigenschaft	*Klasse*	*Property*
Farbe der Tool Bar	`UIToolBar`	`tintColor`
Farbe der Navigation Bar	`UINavigationBar`	`tintColor`
Farbe der selektierten Zelle in Tabellen	`UIView`	`backgroundColor`
Farbe des Zellenseparators in Tabellen	`UITableView`	`separatorColor`
Rand um Grouped Table View	`UITableView`	`backgroundColor`
Farbe der Schrift in Labels	`UILabel`	`textColor`

Damit soll der Abschnitt »Farben« abgeschlossen werden. Wie angekündigt, waren das wirklich nur einfache, grundlegende Anpassungen. Die Auswirkungen sind allerdings beachtlich. Auf eine Abbildung wird an dieser Stelle wegen des Schwarz-Weiß-Drucks des Buchs verzichtet. Sie finden das Bild allerdings online unter der Adresse *www.buch.cd*. Im Internet finden Sie außerdem zahlreiche Tutorials, um Table Views weiter zu modifizieren.

22.4 Schrift

Zur Definition von Schriftart und -größe wird die Klasse `UIFont` verwendet. Sie enthält einige Klassenmethoden, mit deren Hilfe die Schrift erzeugt wird. Die Methode `fontWithName:size:` etwa bekommt den Namen der Schrift sowie die gewünschte Größe übergeben:

```
label.font = [UIFont fontWithName:@"Arial Rounded MT Bold" size:(36.0)];
```

Um die Methode nutzen zu können, muss natürlich bekannt sein, welche Schriftarten auf dem Gerät zur Verfügung stehen. Auch hier hilft die Klasse `UIFont`.

Die Methode `familyNames` gibt die Namen aller auf dem Gerät verfügbaren Schriftfamilien als Array zurück. Innerhalb einer Familie existieren mehrere Schriften, deren Namen man ebenfalls als Array mit `fontNamesForFamilyName:` erhält. Kombiniert man die beiden Methoden wie in folgendem Beispiel, dann erhält man eine Liste aller auf dem Gerät verfügbaren Schriften:

```
NSArray *families = [UIFont familyNames];
for (NSString* family in families)
{
  NSLog(family);
  NSArray *fonts = [UIFont fontNamesForFamilyName:family];
  for (NSString* font in fonts)
  {
```

```
        NSLog(@"    %@",font);

    }
}
```

Auf die komplette Wiedergabe wird hier verzichtet, lediglich einige Zeilen der Ausgabe sind im Folgenden zum besseren Verständnis wiedergegeben:

```
09:28:04.499 chronos[55099:20b] AppleGothic
09:28:04.502 chronos[55099:20b]     AppleGothic
09:28:04.504 chronos[55099:20b] Hiragino Kaku Gothic ProN
09:28:04.506 chronos[55099:20b]     HiraKakuProN-W6
09:28:04.507 chronos[55099:20b]     HiraKakuProN-W3
09:28:04.509 chronos[55099:20b] Arial Unicode MS
09:28:04.509 chronos[55099:20b]     ArialUnicodeMS
09:28:04.511 chronos[55099:20b] Heiti K
09:28:04.513 chronos[55099:20b]     STHeitiK-Medium
09:28:04.514 chronos[55099:20b]     STHeitiK-Light
09:28:04.515 chronos[55099:20b] DB LCD Temp
09:28:04.516 chronos[55099:20b]     DBLCDTempBlack
09:28:04.517 chronos[55099:20b] Helvetica
09:28:04.518 chronos[55099:20b]     Helvetica-Oblique
09:28:04.518 chronos[55099:20b]     Helvetica-BoldOblique
09:28:04.519 chronos[55099:20b]     Helvetica
09:28:04.520 chronos[55099:20b]     Helvetica-Bold
09:28:04.521 chronos[55099:20b] Marker Felt
09:28:04.523 chronos[55099:20b]     MarkerFelt-Thin
09:28:04.524 chronos[55099:20b] Times New Roman
09:28:04.524 chronos[55099:20b]     TimesNewRomanPSMT
09:28:04.526 chronos[55099:20b]     TimesNewRomanPS-BoldMT
09:28:04.526 chronos[55099:20b]     TimesNewRomanPS-BoldItalicMT
09:28:04.527 chronos[55099:20b]     TimesNewRomanPS-ItalicMT
09:28:04.528 chronos[55099:20b] Verdana
09:28:04.529 chronos[55099:20b]     Verdana-Bold
09:28:04.530 chronos[55099:20b]     Verdana-BoldItalic
09:28:04.530 chronos[55099:20b]     Verdana
09:28:04.531 chronos[55099:20b]     Verdana-Italic
```

Bild 22.7: Ein Teil der verfügbaren Schriftarten

Es wäre ganz nützlich zu wissen, wie all die Schriftarten nun aussehen. Vielleicht versuchen Sie sich mal an einer kleinen Anwendung, die in einem Table View die verschiedenen Fonts anzeigt. Eventuell sogar mit Drill-Down von der Familie zum Font?

22.5 Launch Image

Die Akzeptanz für eine lange Ladezeit von Programmen ist auf mobilen Geräten besonders gering. Die App soll möglichst sofort nach dem Tap auf das Start-Icon verfügbar sein. Um das zu gewährleisten, kann ein kleiner Trick genutzt werden.

Beim Start der Anwendung wird vom System eine Grafik namens `Default.png` in den Projekt-Ressourcen gesucht und sofort angezeigt, sofern sie gefunden wird.

Der Trick besteht nun darin, vom ersten Screen der Anwendung einen Screenshot anzufertigen und unter dem Namen `Default.png` abzuspeichern. Beim Start der Anwendung wird zunächst der Screenshot angezeigt, der anschließend durch den echten View ersetzt wird. Der Benutzer hat die Illusion, dass die Anwendung sofort zur Verfü-

gung steht. Bis er anfängt, die App zu benutzen, ist der richtige View (hoffentlich) schon geladen. Die folgende Abbildung zeigt ein Default-Image und den danach angezeigten echten View:

Bild 22.8: Default-Image und View

Ein Screenshot wird durch gleichzeitiges Betätigen der *Home*- und *Ein/Aus*-Taste des iPhones angefertigt. Er kann dann durch *iPhoto* importiert und als `Default.png` den Projekt-Ressourcen zugefügt werden. Diese Variante ist relativ umständlich, soll hier aber dennoch vorgestellt werden weil sie jederzeit, auch unterwegs, die Möglichkeit bietet, Screenshots anzufertigen.

Alternativ dazu erlaubt auch der Xcode Organizer, bei angeschlossenem Gerät einen Screenshot anzufertigen und sogar gleich als Default-Image abzuspeichern. Die Funktionalität findet sich im Reiter *Screenshots* des Organizers, und die entsprechenden Schaltflächen heißen *Capture* bzw. *Save as Default Image...*

Bild 22.9: Screenshots-Bereich im Organizer

Für Anwendungen, die gleich auf dem ersten Screen dynamische Daten anzeigen, ist ein kompletter Screenshot natürlich wenig sinnvoll. Die *Wetter*-App ist ein gutes Beispiel dafür. Zunächst das Wetter vom Tag des Screenshots anzuzeigen, um dann auf das aktuelle Wetter umzublenden, kann je nach Wetterlage Freudentränen oder schiere Verzweiflung auslösen. Was ist zu tun? Falls Ihr Gerät langsam genug ist, können Sie beim Start der *Wetter*-App erkennen, dass zunächst ein Screenshot ohne irgendwelche Texte angezeigt wird. Das gibt dem Benutzer die sofortige, wichtige Rückmeldung, dass sein Startbefehl angenommen wurde. Danach wird der echte View mit den sich dynamisch ändernden Wochentagen eingeblendet.

Die *iPhone Human Interface Guidelines* weisen übrigens explizit darauf hin, dass das Launch Image nicht für einen Splash-Screen oder ein About-Fenster benutzt werden soll. Das ist ohnehin keine gute Idee, weil die Ladezeit für den echten View abhängig von der Hardware variiert und das About-Fenster deshalb möglicherweise viel zu kurz angezeigt wird.

Teil 4 – Die Auslieferung

Die Zeiterfassung ist fertig. Im letzten Teil des Buchs soll es darum gehen, wie das Werk getestet und anschließend in den App Store eingestellt wird. Das Buch schließt mit einem »weichen« Kapitel über das Geschäftsmodell für iPhone-Entwickler.

23 Unit Tests

Unit Tests sind als Hilfsmittel aus der Softwareentwicklung nicht mehr wegzudenken und haben in allen Programmiersprachen Einzug gehalten. Man muss kein Anhänger von *Extreme Programming* [BEC03] und der Test-First-Philosophie sein, um anzuerkennen, dass die Tests die Qualität der Software ungemein steigern. Erst eine ordentliche Testabdeckung ermöglicht *Refactoring* [FOW00], das ständige häppchenweise Verbessern der Software, und führt damit zu lebendigem, wartbarem Code. Die früher so gefürchteten Seiteneffekte (Fehler, die an ganz anderer Stelle auftreten als an der, wo der Code geändert wird) lassen sich damit unter Kontrolle bringen.

Dank dem Open-Source-Projekt *SenTestingKit* (auch bekannt als *OCUnit*) stehen dem iPhone/iPad-Entwickler Unit Tests in zwei Varianten zur Verfügung.

23.1 Was soll getestet werden?

Vor dem Schreiben der eigentlichen Tests sollten Sie sich Gedanken darüber machen, was getestet werden soll. Gar nicht zu testen wäre sehr riskant. Selbst der erfahrenste Entwickler kann bei seiner Arbeit die ungewollten Seiteneffekte nicht überblicken. Dies trifft insbesondere für Änderungen an fremdem Code oder bei Wartungsarbeiten lange Zeit nach der eigentlichen Entwicklung zu.

Alle Methoden zu testen ist ebenso problematisch. Eine hundertprozentige Testabdeckung ist zumindest bei Unit Tests ein teures und vollkommen unnötiges Unterfangen. Es macht keinen Sinn, triviale Methoden zu testen, nur um sich an einer hohen Testabdeckung zu erfreuen. Unit Tests für UI-Komponenten sind mit den später vorgestellten *Application Tests* zwar grundsätzlich möglich, aber meist steht der dafür notwendige Aufwand in keinem vernünftigen Verhältnis zum Nutzen.

Bei allem Bekenntnis zu qualitativ hochwertigerer Software sollten die Tests doch vorwiegend für gut zu testende und vor allen Dingen potenziell fehlerträchtige Bereiche implementiert werden. Das sind weniger die bisher besprochenen eher trivialen Callback-Methoden für die GUI-Steuerung. Potenzielle Kandidaten sollten Sie eher im Bereich Ihrer Geschäftslogik suchen. Überall dort, wo umfangreiche Berechnungen laufen oder komplizierte Algorithmen ihr Werk verrichten, machen sich Unit Tests ganz besonders bezahlt.

23.2 Logische Tests

Logische Tests werden unabhängig von der Anwendung ausgeführt, in der die zu testenden Methoden normalerweise laufen. Wie Sie sich bestimmt vorstellen können, hat diese Einschränkung Auswirkung auf die zu testenden Codebestandteile. Logische Tests bieten sich vor allen Dingen zum Testen von Bibliotheken, Frameworks und Backend-Komponenten (also des Models im Sinne des MVC-Patterns) an. Schlecht bis gar nicht testbar sind alle UI-Komponenten.

Einfachstes Beispiel für logische Tests sind Methoden, die mit übergebenen Parametern ein Ergebnis berechnen. Der Testfall ruft die Methode mit verschiedenen Werten auf und überprüft, ob der Rückgabewert dem erwarteten Ergebnis entspricht. Eins und eins muss zwei ergeben, ganz egal in welcher Anwendung der Code ausgeführt werden soll.

Die eigentliche Anwendung läuft bei der Ausführung des Codes nicht, weder auf dem Gerät noch auf dem Simulator.

Zum Ausführen der Tests wird ein neues Target benötigt. Ein Rechtsklick auf *Targets* in der *Group & Files*-Ansicht öffnet das Kontextmenü, in dem via *Add > New Target...* neue Targets angelegt werden können. Als Vorlage für das Target wird *Unit Test Bundle* ausgewählt. Dem Target werden im Folgenden Testsuiten zugefügt, deren Tests beim Build des Targets ausgeführt werden. Als Targetname wird *LogicTests* vergeben.

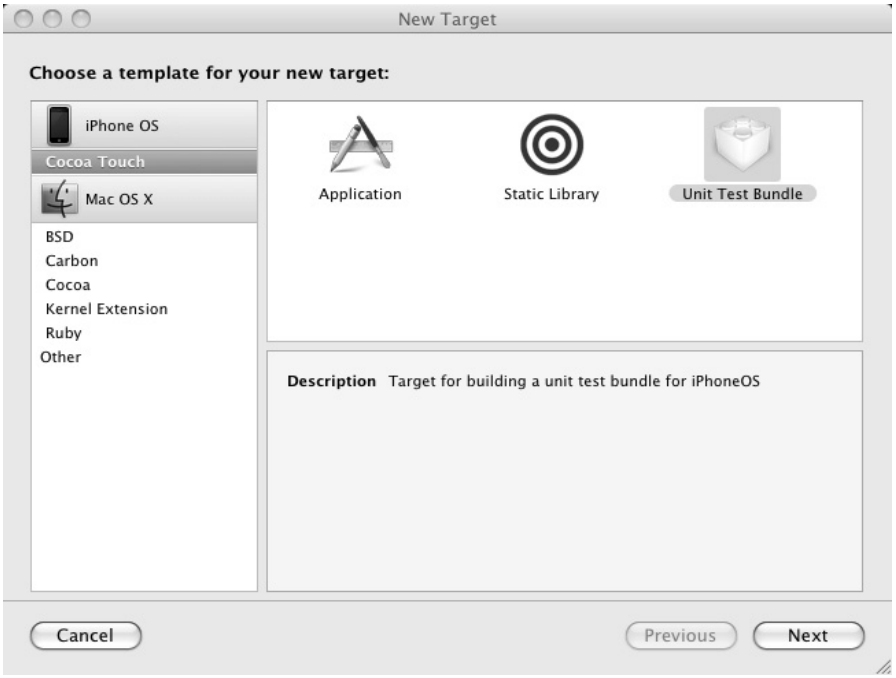


Bild 23.1: Erstellen eines Unit Test Bundles

Das neue Target taucht nach erfolgreichem Anlegen in *Groups & Files* unter *Targets* auf und muss zum Ausführen des Tests in der *Overview*-Auswahlliste in der Tool Bar als *Active Target* ausgewählt werden.

Als Nächstes wird eine neue Gruppe namens *Tests* in *Groups & Files* angelegt. Der Gruppe können nun Testsuiten zugefügt werden. Testsuiten sind Klassen mit mehreren Testmethoden. Die Testklassen müssen das *SenTestingKit* importieren und von *SenTestCase* erben. Zum Glück gibt es eine Vorlage namens *Objective-C test case class*, die diese Anforderungen schon erfüllt. Wichtig beim Anlegen der Testklasse ist die Auswahl des zuvor erstellten Targets.

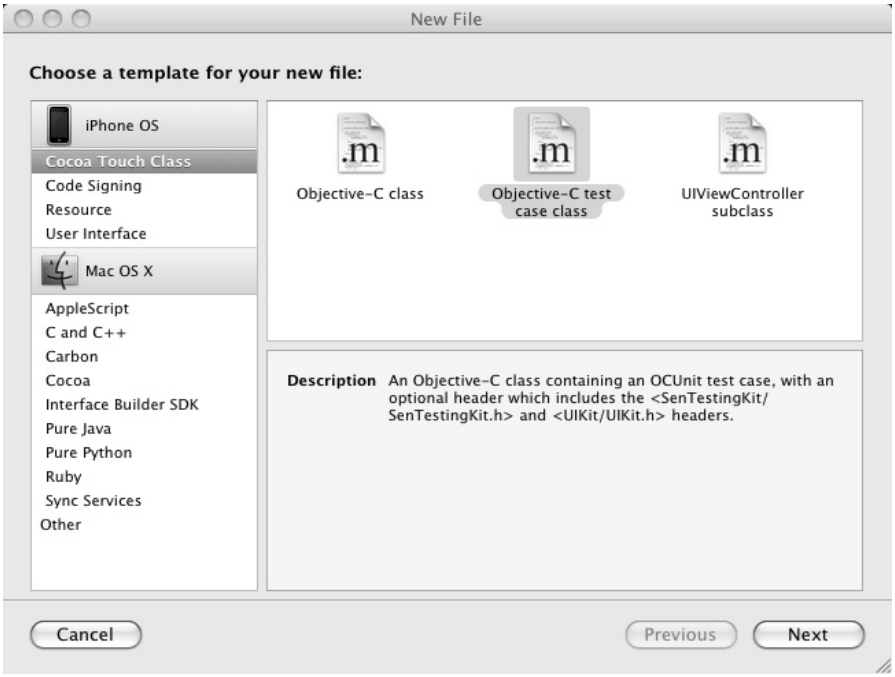


Bild 23.2: Erzeugen einer Testklasse

Der folgende Codeabschnitt zeigt die Header-Datei der erzeugten Testklasse. Es empfiehlt sich, ein wenig aufzuräumen, da das Template sowohl logische als auch Applikationstests in einer Klasse unterbringt. Alternativ kann auch einfach die Konstante `USE_APPLICATION_UNIT_TEST` auf den Wert 0 gesetzt werden. Dann wird der für logische Tests generierte `else`-Zweig mit der Methode `testMath` durchlaufen.

```
#define USE_APPLICATION_UNIT_TEST 1

#import <SenTestingKit/SenTestingKit.h>
#import <UIKit/UIKit.h>
//#import "application_headers" as required
```

```
@interface ChronosAppTests : SenTestCase {
}

#if USE_APPLICATION_UNIT_TEST
- (void) testAppDelegate;       // simple test on application
#else
- (void) testMath;              // simple standalone test
#endif

@end
```

Die Implementierung der Testklasse sieht (nach dem Entfernen von nicht benötigtem Code) folgendermaßen aus:

```
#import "LogicTests.h"

@implementation LogicTests

- (void) testMath {
    STAssertTrue((1+1)==2, @"Compiler isn't feeling well today :-(" );
}

@end
```

Testfall-Methoden müssen mit `test` beginnen und Instanzmethoden sein. Sie dürfen keine Parameter enthalten und der Rückgabetyp muss `void` sein.

Der generierte Testfall `testMath` zeigt die grundsätzliche Vorgehensweise. Durch das Makro `STAssertTrue` wird überprüft, ob der danach angegebene Ausdruck `true` ergibt. Im Fall `1+1==2` ist das nicht weiter spannend. Die Durchführung bleibt aber die gleiche, auch im Fall komplexer Methodenresultate.

Den Test kann man durch einen einfachen Build des zuvor hoffentlich umgestellten *Active Target* laufen lassen. Als *Active SDK* muss der Simulator ausgewählt sein. Verläuft der Test erfolgreich, wird die Meldung *Build succeeded* in der Statuszeile angezeigt.

Berechnet sich ein Ausdruck dagegen zu `false`, wird eine Fehlermeldung erzeugt:

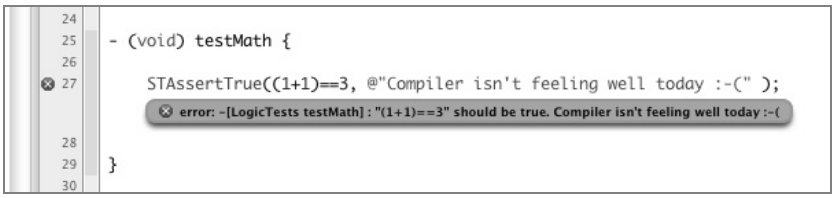


Bild 23.3: Fehlgeschlagener Test

Neben `STAssertTrue` stehen einige weitere Makros zur Validierung des Testergebnisses zur Verfügung:

- `STAssertEquals`, `STAssertEqualObjects`, `STAssertEqualsWithAccuracy`
- `STAssertNil`, `STAssertNotNil`
- `STAssertTrue`, `STAssertFalse`
- `STAssertThrows`, `STAssertThrowsSpecific`

Die technischen Voraussetzungen sollten damit geklärt sein. Aufgabe des Entwicklers ist es nun, verschiedene Testmethoden zu implementieren, die statt des Ausdrucks 1+1 das Ergebnis eines Methodenaufrufs auf Korrektheit überprüfen. Dazu wird eine Instanz der betroffenen Klasse erzeugt und die zu testende Methode mit den entsprechenden Parametern aufgerufen.

Ein ganz praktisches Hilfsmittel bei dieser Aufgabe sind die Methoden `setUp` und `tearDown`, die automatisch vor bzw. nach jedem Testfall aufgerufen werden. Damit lässt sich die Instanz der zu testenden Klasse bequem erzeugen bzw. nach durchlaufenem Test wieder beseitigen. Das folgende Listing zeigt ein Beispiel dafür. Es wird die Methode `serializeObject` der im Kapitel »Zugriff aufs Internet« vorgestellten Klasse `CJSONSerializer` getestet:

```
#import "LogicTests.h"
#import "CJSONSerializer.h"

@implementation LogicTests

- (void) setUp {
   serializer = [[CJSONSerializer alloc]init];
}

- (void) testCJSONSerializer {

   NSMutableArray *arrayDict = [[NSMutableArray alloc] init];
   NSArray *keys = [NSArray arrayWithObjects:@"projektkey", @"leistungkey",
     @"datumkey", @"dauerkey", nil];
   NSArray *objects = [NSArray arrayWithObjects:@"projektvalue",
     @"leistungvalue", @"datumvalue", @"dauervalue", nil];
   NSDictionary *dictionary = [NSDictionary dictionaryWithObjects:objects
     forKeys:keys];
   [arrayDict addObject:dictionary];

   NSString *jsonString = [serializer serializeObject:arrayDict];
    STAssertEqualObjects(jsonString,
@"[{\"dauerkey\":\"dauervalue\",\"leistungkey\":\"leistungvalue\",\"datumkey
\":\"datumvalue\",\"projektkey\":\"projektvalue\"}]",
     @"But was :%@",  jsonString);
}
```

```
- (void) tearDown {
   [serializer release];
}

@end
```

Wichtig beim Schreiben der Tests ist es, immer auch den Negativfall durchzuspielen, also zu prüfen, ob ein Test, der scheitern sollte, auch wirklich scheitert. Danach wird der Vergleichswert auf das tatsächlich erwartete richtige Ergebnis gesetzt und der Test sollte erfolgreich durchlaufen. Hält man sich nicht an diese Vorgehensweise, kann man leicht durch Fehler im Test selber ein (falsches) positives Ergebnis vorgetäuscht bekommen.

Falls Sie beim Erstellen der Tests feststellen, dass sich der Code in der momentanen Form nur sehr schwer testen lässt, etwa weil keine sauberen Schnittstellen existieren und alles zu verwoben ist, sollten Sie vielleicht doch einen Blick auf den Test-First-Ansatz werfen. Der Test wird dabei vor der zu testenden Methode geschrieben, was den Entwickler zwingt, die Methode testbar zu implementieren. Das Resultat ist insgesamt besser wartbarer Code.

23.3 Application Tests

Application Tests werden bei laufender Anwendung direkt auf dem Gerät (nicht im Simulator!) ausgeführt. Mit dieser Testart, die eher einem Integrationstest entspricht, können auch View und Controller getestet werden.

Die Einbindung ist ein klein wenig komplizierter als die der logischen Tests. Zunächst wird eine Kopie des Targets, das die Anwendung baut, erstellt. Dazu wird die Funktion `Duplicate` im Kontextmenü benutzt. Die Kopie wird in unserem Beispielprojekt in `chronosTesting` umbenannt, um die Verwendung anzudeuten.

Anschließend wird wiederum ein *Unit Test Bundle* benötigt. Als Name wird `chronosTests` vergeben. Das Test Bundle wird die Testklassen enthalten und soll vor `chronosTesting` kompiliert werden. Um diese Abhängigkeit zu bewirken, wird per Drag & Drop `chronosTests` in das `chronosTesting`-Target gezogen.

Die Targets sehen nun folgendermaßen aus:

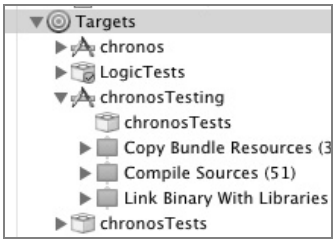


Bild 23.4: Targets mit Application Tests

Als Nächstes ist der Ordner *Products* des Projekts zu öffnen. Von hier wird das Produkt `chronosTests.octest` in die Phase *Copy Bundle Resources* des Targets `chronosTesting` gezogen:

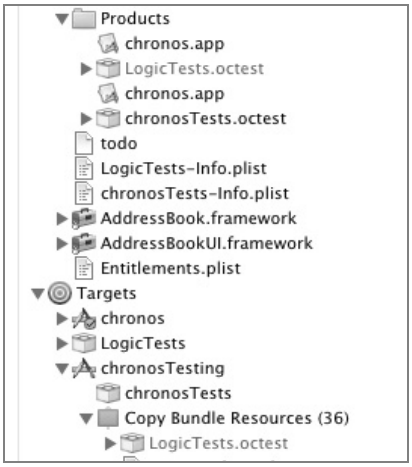


Bild 23.5: Targets mit `LogicTests.octest`

Danach wird analog zu den logischen Tests eine Testklasse mithilfe der Vorlage *Unit Test Case Class* erzeugt.

Application Tests haben, wie erwähnt, Zugriff auf die laufende Anwendung. Der Test im `if`-Zweig der Vorlage nutzt dieses Feature, um zu prüfen, ob das `ApplicationDelegate` ungleich `nil` ist:

```
- (void) testAppDelegate {
   id yourApplicationDelegate = [[UIApplication sharedApplication]
delegate];
   STAssertNotNil(yourApplicationDelegate, @"UIApplication failed to find
the
     AppDelegate");
}
```

Vom `ApplicationDelegate` aus kann auf weitere Testkandidaten wie Controller und deren Views zugegriffen werden.

Ein Start des `chronosTesting`-Targets mit *Build & Go* auf dem Gerät führt zu folgender Ausgabe auf der Konsole:

```
Test Suite 'All tests' started
Test Suite '/Developer/Library/Frameworks/SenTestingKit.framework(Tests)'
started Test Suite 'SenInterfaceTestCase' started
Test Suite 'SenInterfaceTestCase' finished
Executed 0 tests, with 0 failures (0 unexpected) in 0.000 (0.005) seconds

Test Suite '/Developer/Library/Frameworks/SenTestingKit.framework(Tests)'
finished.
```

```
Executed 0 tests, with 0 failures (0 unexpected) in 0.000 (0.017) seconds

Test Suite '/var/mobile/Applications/7B112640-EB6F-44E5-871D-
5CAD23DF0919/chronos.app/chronosTests.octest(Tests)'
Test Suite 'ChronosAppTests' started
Test Case '-[ChronosAppTests testAppDelegate]' passed (0.000 seconds).
Test Suite 'ChronosAppTests'.
Executed 1 test, with 0 failures (0 unexpected) in 0.000 (0.007) seconds

Test Suite '/var/mobile/Applications/7B112640-EB6F-44E5-871D-
5CAD23DF0919/chronos.app/chronosTests.octest(Tests).
Executed 1 test, with 0 failures (0 unexpected) in 0.000 (0.019) seconds

Test Suite 'All tests' finished.
Executed 1 test, with 0 failures (0 unexpected) in 0.000 (0.063) seconds
```

Wie man der Ausgabe entnehmen kann, wird der Test erfolgreich ausgeführt.

Wer sich tiefer mit dem Thema Unit Tests beschäftigen möchte, sollte sich auch *GTM* [URL-GTM] und *GHUnit* [URL-GHUNIT] ansehen. Zur Verwendung muss zwar externer Code eingebunden werden, dafür sind die Frameworks einfacher zu benutzen und zu konfigurieren.

24 Beta-Test

Alles bereit für die Auslieferung in den App Store? Licht aus, Spot an. Ladies and Gentlemen, we proudly present –

Stopp!!!

Bevor Ihr Meisterwerk endlich die große Showbühne erblickt, sollten unbedingt einige Testpersonen einen kritischen Blick auf die App geworfen haben. Entwickler schauen gerne mit der rosaroten Brille auf ihr »Baby« und meiden fehlerträchtige Funktionen. Der Test durch Freunde, Bekannte, Arbeitskollegen oder Kunden bringt die ganze grässliche Wahrheit ans Licht.

Neben den von Ihnen wohlwollend übersehenen Unschönheiten treten beim Betatest nun auch Probleme mit verschiedenen Geräteversionen und Konfigurationen auf. Falls die App in mehreren Sprachen angeboten werden soll, haben die entsprechenden Testpersonen zum Thema Internationalisierung womöglich auch noch die ein oder andere Anmerkung. Vor der Auslieferung gibt es also mit hoher Wahrscheinlichkeit doch noch einiges zu tun.

24.1 Herausfinden der Geräte-ID

Für den Beta-Test müssen Sie Ihre App schon vor der Verfügbarkeit im App Store auf die Geräte anderer Personen bringen. Und das möglichst, ohne dass die iPhones der Tester extra mit Ihrem Rechner verbunden werden müssen. Dabei hilft Apples Ad-hoc-Profil.

Zur Erstellung eines *Ad Hoc Provisioning Profile* benötigt man, wie auch beim Entwicklungsprofil, die IDs der Geräte, auf denen getestet werden soll.

Wie bereits im Kapitel »Projektstart« erläutert, kann man die ID mithilfe von *iTunes* herausfinden. Dazu wird bei angeschlossenem Gerät in der Baumansicht auf der linken Seite des iTunes-Fensters das betreffende iPhone oder iPad unter *Geräte* ausgewählt. Im Reiter *Übersicht* klickt man dann auf *Seriennummer*. Die Seriennummer verschwindet daraufhin und macht der gesuchten ID Platz.

Die ebenfalls besprochene Alternative, die ID mithilfe des Xcode Organizers herauszufinden, ist für den Beta-Test nicht sonderlich hilfreich. Viele Testpersonen werden Xcode gar nicht installiert haben und womöglich auch nicht installieren können (weil sie einen Windows-Rechner nutzen).

Allerdings gibt es für Beta-Tests eine ganz hilfreiche App namens *BetaHelper*, die die ID ebenfalls ermittelt. Die Anwendung ist kostenlos im App Store erhältlich. Die folgende Abbildung zeigt einen Screenshot des Tools:

Bild 24.1: BetaHelper

Sie können Ihre Tester also entweder mit einer Beschreibung versorgen, die erklärt, wie man iTunes die ID entlockt. Oder aber Sie bitten sie einfach, die App zu installieren und über die entsprechende Funktionalität die Mail mit der ID an Sie zu versenden.

Sind die IDs der Testgeräte bekannt, werden sie, wie im Kapitel »Projektstart« beschrieben, unter *Devices* im *Developer Program Portal* angelegt.

24.2 Distributionszertifikat

Unglücklicherweise muss für die Ad-hoc-Verteilung auch ein weiteres Zertifikat, das Distributionszertifikat, angelegt werden. Dazu wird, wie ebenfalls im Kapitel »Projektstart« beschrieben, mithilfe der Mac-Anwendung *Schlüsselbundverwaltung* (Menüpunkt *Schlüsselpunktverwaltung > Zertifikatsassistent > Zertifikat einer Zertifizierungsinstanz anfordern*) ein *Certificate Signing Request* erstellt. Die Anfrage wird auf der Festplatte gespeichert und anschließend im *Developer Program Portal* unter *Certifcates* im Reiter *Distribution* (!) hochgeladen.

Nach dem Klick auf *Approve* und einer kurzen Wartezeit kann das fertige Zertifikat heruntergeladen werden. Ein Doppelklick darauf öffnet den Assistenten der Schlüsselbundverwaltung zum Zufügen neuer Zertifikate. Ist dies geschehen, sollte sich ein Eintrag *iPhone Distribution: {Ihr Name}* in der Liste der Zertifikate im ausgewählten Schlüsselbund befinden:

Bild 24.2: Zertifikate im Schlüsselbund

Das Zertifikat wird im Folgenden verwendet, um das Ad-hoc-Profil zu erstellen.

24.3 Ad-hoc-Profil

Sind die Geräte eingetragen und das Zertifikat erstellt, wird als Nächstes ein *Ad Hoc Distribution Provisioning Profile* angelegt. Es handelt sich dabei um eine Art Erlaubnis, die App auf den Geräten der Tester laufen zu lassen. Das Pofil wird im *Developer Program Portal* unter *Provisioning* angelegt. Diesmal ist allerdings auch hier der Reiter *Distribution* auszuwählen. Auf der Seite mit der Liste der vorhandenen Profile findet sich der Button *New Profile* zum Anlegen neuer Profile. Als *Distribution Method* wird *Ad Hoc* ausgewählt und dann ein sinnvoller Name für das Profil vergeben (etwa *Chronos Ad Hoc Distribution Profile*). Außerdem werden die App ID (*chronos*) und die gewünschten Geräte selektiert:

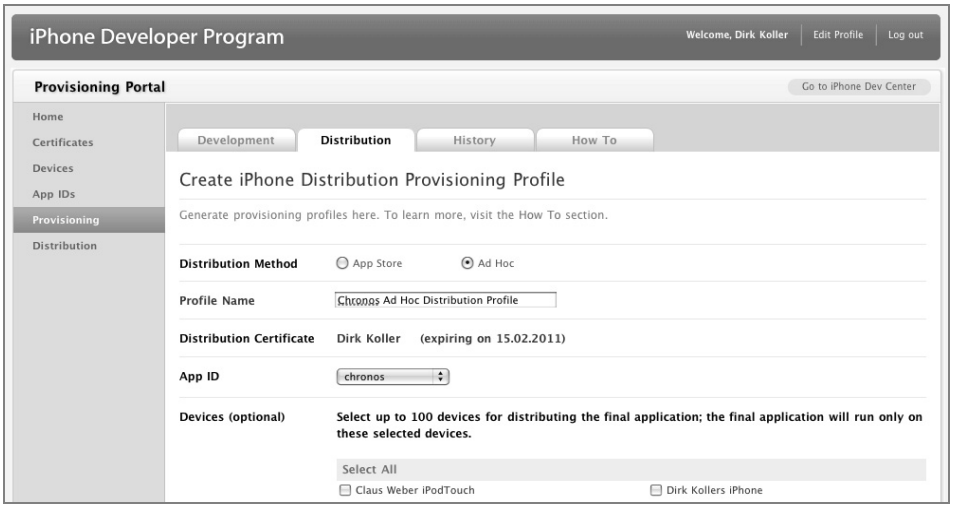


Bild 24.3: Erzeugen des Ad-hoc-Profils

Nach dem Absenden des Formulars und einer kurzen Wartezeit ist das Profil fertig und steht zum Download bereit:

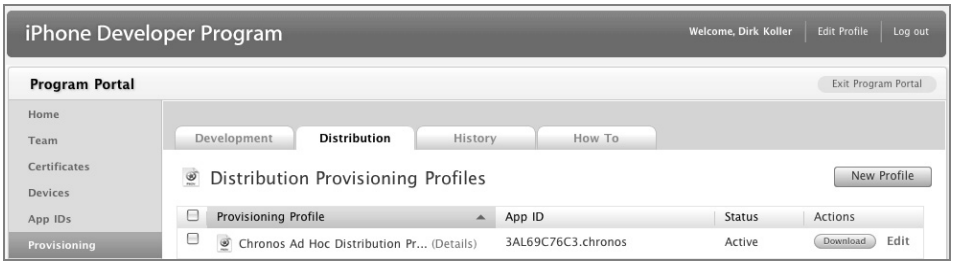


Bild 24.4: Das fertige Profil

Wie das Entwicklungsprofil muss auch das Ad-hoc-Profil nach dem Download in Xcode registriert werden. Dazu öffnet man in Xcode den Organizer (*Window > Organizer*) und zieht das Profil direkt aus dem Finder auf *Provisioning Profiles* (unter *IPHONE DEVELOPMENT*). Anders als das Entwicklungsprofil braucht das Ad-hoc-Profil nicht auf ein eventuell angeschlossenes Gerät gezogen zu werden. Das Profil soll ja auf den Geräten der Tester und nicht auf dem eigenen Gerät installiert werden.

24.4 Erzeugen des Build

Die Versionsnummer in `info.plist` sollte für jeden Ad-hoc-Build erhöht werden. Zum einen kann ein zweiter Build mit gleicher Versionsnummer bei der Installation Probleme bereiten, zum anderen ist es für Sie wichtig, den Überblick zu behalten, mit welcher Version die Tester arbeiten.

Außerdem ist es aus letzterem Grund natürlich sinnvoll, bei jedem Ad-hoc-Build einen *Tag* in der Versionsverwaltung zu setzen. Während des Tests arbeiten Sie womöglich weiter am Code, müssen bei gemeldeten Fehlern aber den Codestand bei Erzeugen des Builds kennen. Zum Erzeugen eines Tags in Subversion wird der aktuelle Stand unter *trunk* einfach unter einem neuen Namen in das Verzeichnis *tags* kopiert. Der dazu notwendige Befehl lautet (aus dem Verzeichnis *chronos* ausgeführt):

```
svn copy trunk/ tags/NamesDesTags
```

Für den Beta-Build wird in Xcode eine neue Konfiguration angelegt. Dazu wird in den Projektinformationen im Reiter *Configurations* die *Release*-Konfiguration mit dem Button *Duplicate* kopiert und die Kopie in *Distribution* umbenannt. Danach wählt man im Reiter *Build* die neue Konfiguration in der Auswahlliste links oben aus.

Als Nächstes wird das Setting *Code Signing Identity* für die neue Konfiguration eingestellt. Hier ist das neue Profil mit dem Namen *iPhone Distribution:{Ihr Name}* auszuwählen.

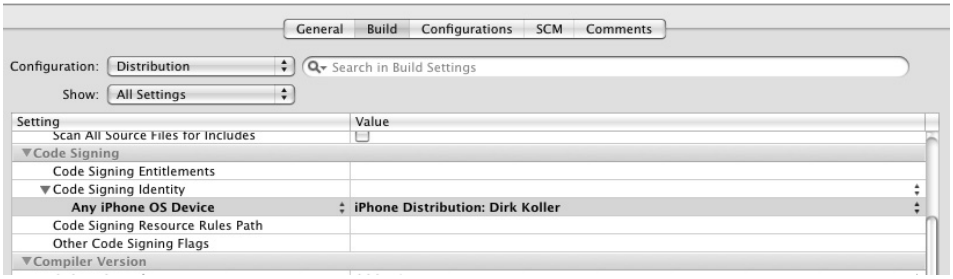


Bild 24.5: Auswahl der Code Signing Entity in den Projektinformationen

Man sollte meinen, dass damit nun endlich die Vorbereitungen und Konfigurationen abgeschlossen sind und der Build erstellt werden kann. Weit gefehlt. Für eine Ad-hoc-Verteilung muss dem Projekt noch eine *Entitlement*-Datei zugefügt werden. Darin enthalten sind Properties, die den Zugriff Ihrer App auf iPhone-OS-Features regeln.

Die Datei wird im Menü über *New File* > *iPhone OS* > *Code Signing* > *Entitlements* angelegt:

Bild 24.6: Erzeugen von `Entitlements.plist`

Als Name der Datei wird `Entitlements.plist` gewählt.

Wenn man die neue Datei `Entitlement.plist` in Xcodes *Groups & Files* auswählt, wird die einzige enthaltene Property namens `get-task-allow` im Editor angezeigt:

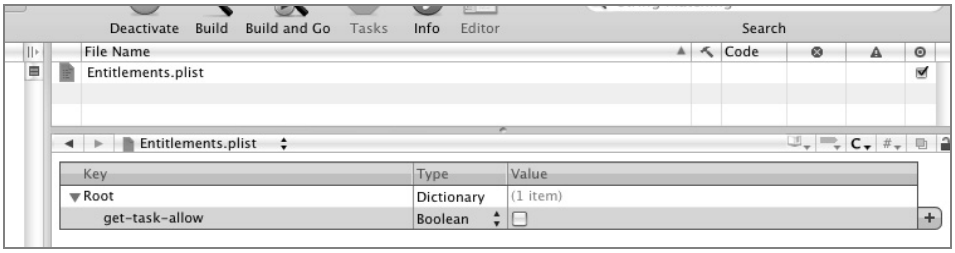


Bild 24.7: Property `get-task-allow` in `Entitlement.plist`

Hier ist die Checkbox in der Spalte *Value* zu deselektieren (die Property wird also auf den Wert `false` gesetzt).

Schließlich muss der Name des Entitlements-Files (inklusive Endung) in den Projektinformationen für die Konfiguration *Distribution* unter *Code Signing Entitlements* eingetragen werden:

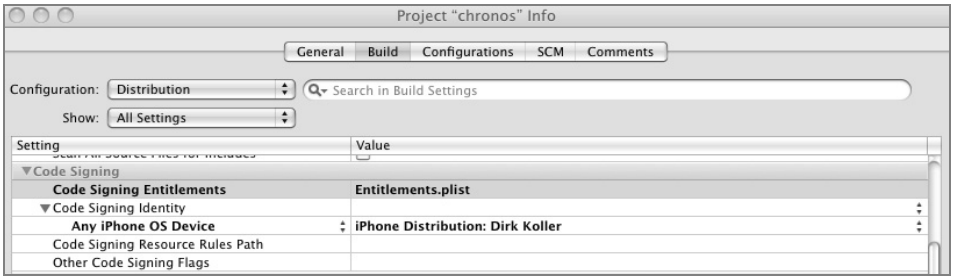


Bild 24.8: Setzen der Code Signing Entitlements

Das war's nun aber wirklich, der Build kann erzeugt werden. Die Overview-Auswahlliste wird dazu auf die gewünschte SDK-Version des Geräts (nicht des Simulators!) eingestellt. Als Konfiguration wird *Distribution* ausgewählt.

Bild 24.9: Einstellungen für den Build

Nach einem letzten Clean wird der Build gestartet. Während der Prozedur erfolgt vor dem Signieren eine Abfrage, ob der Zugriff auf das Zertifikat im Schlüsselbund erlaubt werden soll. Diese ist positiv zu beantworten.

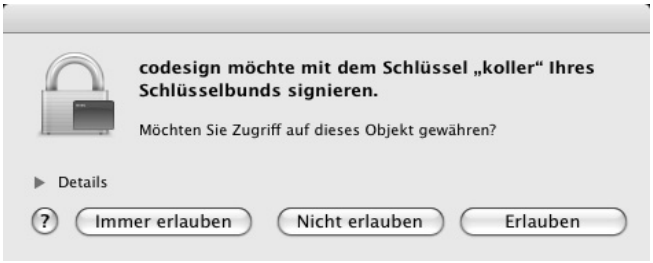


Bild 24.10: Die Signiererlaubnis einholen

Hat alles geklappt, kann der fertige Build lokalisiert werden. Dazu wird in *Groups & Files* unterhalb des Projekts (also *chronos*) der Ordner *Products* geöffnet. Hier findet sich die Datei `chronos.app`. Durch einen Rechtsklick darauf und die anschließende Auswahl von *Reveal in Finder* im Kontextmenü gelangt man zu der Datei im Dateisystem.

24.5 Verteilen von App und Ad-hoc-Profil

Der Build ist erstellt und kann an die Benutzer ausgeliefert werden. Für Mac-Benutzer kann direkt die Datei mit der Endung `.app` verwendet werden.

Für Windows-Tester ist die Sache leider komplizierter. Hier wird ein File mit der Endung `.ipa` benötigt. Um es zu erstellen, kopieren Sie das `.app`-File in ein neues Directory namens *Payload* und komprimieren (vulgo zippen) dieses. Dazu darf aber nicht die *Komprimieren*-Option aus dem Finder benutzt werden.

Relativ einfach geht das Zippen zum Beispiel von der Konsole:

```
mkdir Payload
cp -rp chronos.app Payload/
zip -r chronos.ipa Payload
```

Das `.app`- oder `.ipa`-File wird zusammen mit dem Ad-hoc-Profil am einfachsten per Mail an die Testpersonen verschickt.

Nun stellt sich die Frage, wie die Tester ohne Xcode die App zum Laufen bekommen. Des Rätsels Lösung heißt iTunes. Bei angeschlossenem Gerät wird das `.app`- bzw. `.ipa`-File aus dem Finder bzw. Explorer per Drag & Drop in den Bereich *Programme* der Mediathek von iTunes gezogen. Dort sollte die App nach einem kleinen Moment dann auch angezeigt werden. Mit dem Profil wird analog verfahren.

Bild 24.11: Die App in der Mediathek

Die App wird ebenfalls im Reiter *Programme* bei ausgewähltem iPhone (unter Geräte) angezeigt. Die Checkbox davor muss selektiert sein:

Bild 24.12: Die App ist markiert für das Synchronisieren.

Damit App und Profile auch auf das iPhone kommen, muss noch synchronisiert werden. Danach empfiehlt es sich auf jeden Fall, unter *Allgemein* > *Einstellungen* > *Profile* auf dem Gerät nachzusehen, ob das Profil angekommen ist. Die App sollte auf dem Home-Screen sichtbar sein und kann gestartet werden.

Auch hier noch mal der Hinweis: Der oben besprochene Prozess ist sehr komplex und damit fehleranfällig. Lesen Sie den Text sehr genau und kontrollieren Sie möglichst nach jedem Schritt, ob er erfolgreich war.

24.6 Erfassen der Fehler

Wenn die Beta-Tester die App installiert haben, sollten Sie für Feedback gerüstet sein. Bei kleinen Apps mit wenigen Testern mögen E-Mails der Testpersonen zur Rückmeldung genügen. Vielleicht informieren Sie die betroffenen Personen vorher, welche Informationen eine Fehlermeldung beinhalten sollte. Dadurch kann sichergestellt werden, dass das Feedback auch verwertbar ist. Denkbar sind zum Beispiel:

- Art des Geräts (iPhone, iPod touch, iPad)
- Version des Geräts (z. B. iPhone Classic, 3G, 3G S)
- Version des SDK

- Version der App
- Fehlerbeschreibung
- Screenshots
- Schritte zur Nachstellung des Fehlers

Trotz aller Bemühungen trägt diese Lösung nicht sonderlich weit. E-Mails sind kein gutes Instrument, um Fehler zu verwalten.

Zum Glück gibt es eine auf diesen Anwendungsfall spezialisierte Gattung von Programmen, die Bug-Tracking-Systeme. Sie erlauben das komfortable Erfassen und Verwalten von Fehlern. Zusätzlich verfügen sie über eine ganze Reihe praktischer Features wie etwa ausführliche Reports oder E-Mail-Benachrichtigungen. Namhafte kostenlose Vertreter dieser Spezies sind beispielsweise *Mantis* [URL-MANTIS] oder *Bugzilla* [URL-BUGZILLA]. Eine sehr innovative, allerdings kostenpflichtige Lösung ist *FogBugz* [URL-FOGBUGZ]. Manche dieser Systeme werden auch als SaaS (Software as a Service) angeboten, sodass keine Installation auf eigenen Servern notwendig ist.

Der Platz an dieser Stelle reicht nicht, das Thema weiter zu vertiefen. Falls Sie planen, kommerzielle Software für die iPhone-Plattform zu entwickeln, sollten Sie sich allerdings unbedingt eines der Produkte näher ansehen. Der geringe Aufwand für die Einarbeitung wird durch die vielen Vorteile bei Weitem aufgewogen.

24.7 Crash Logs

Oft genug werden die Fehler, die bei den Testern auftreten, nicht reproduzierbar sein. Ein wichtiges Hilfsmittel für diesen Fall sind die *Crash Logs*, die beim Absturz der App auf dem Gerät gespeichert werden. Wie erwähnt, sind die Logfiles im Xcode Organizer nach dem Anschluss des Geräts verfügbar.

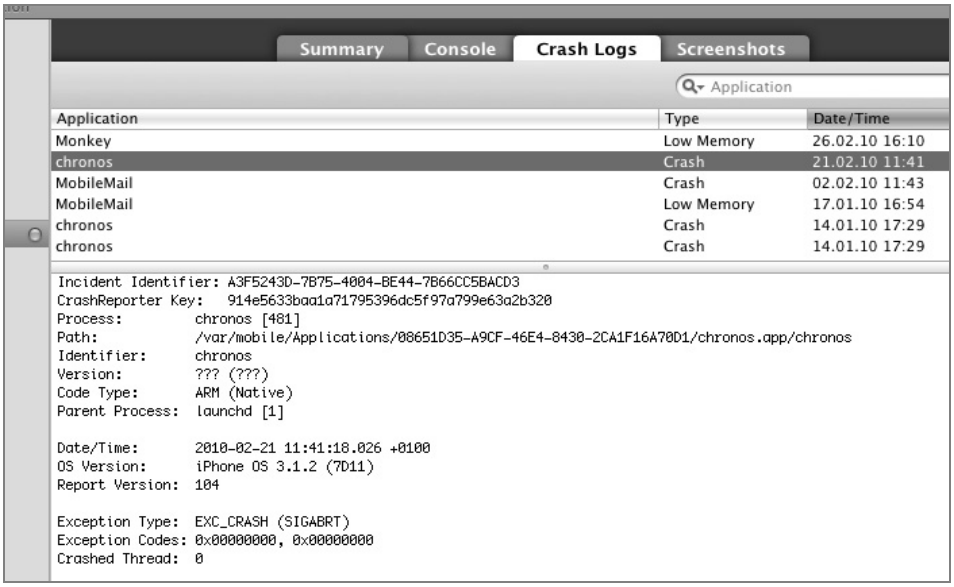


Bild 24.13: Crash Logs im Organizer

Für Tester, die Xcode nicht verwenden, sind die Logs erfreulicherweise auch zugänglich. Man findet sie auf dem Mac unter *~/Library/Logs/CrashReporter/MobileDevice* und unter Windows XP unter *C:\Documents and Settings\<USERNAME>\Application Data\Apple computer\Logs\CrashReporter*, nachdem man das Gerät angeschlossen hat.

Es ist hilfreich, sich bei auftretenden Fehlern von den Testern den Inhalt dieses Verzeichnisses zukommen zu lassen.

25 Auslieferung in den App Store

Der große Moment ist gekommen, die App ist von den Beta-Testern für gut oder zumindest fehlerfrei befunden worden und kann den Weg in den App Store antreten. Nach ein paar Vorbereitungen wird der entscheidende Build erstellt und das Binär-File in den App Store ausgeliefert.

25.1 Letzte Arbeiten

Bevor es wirklich losgehen kann, sind noch ein paar finale Arbeiten zu erledigen.

25.1.1 Benötigte Hardware

Die iPhone-Plattform besteht inzwischen aus einer ganzen Reihe von Geräten mit sehr unterschiedlichen Leistungsmerkmalen. Mit dem iPod touch kann beispielsweise nicht telefoniert werden, die erste iPhone-Generation verfügt über kein GPS usw.

Um Missverständnisse und Enttäuschungen beim Erwerb einer App zu vermeiden, können seit dem iPhone SDK 3.0 in der Datei `info.plist` unbedingt benötigte Hardwaremerkmale für eine App aufgeführt werden. Die Liste hat zwei Auswirkungen: Im App Store kann die betroffene App erstens nur von Geräten geladen werden, die die Anforderungen erfüllen, und zweitens auch nur auf solchen Geräten gestartet werden.

Der Key für `info.plist` heißt `UIRequiredDeviceCapabilities`. Nach dem Hinzufügen des Schlüssels in die Property-Liste wird per Rechtsklick in das *Value*-Feld der *Value Type* auf *Array* gesetzt. Anschließend können über den Button mit dem Listen-Symbol (siehe folgende Abbildung) dem Array Elemente zugefügt werden. Ist eine Eigenschaft im Array aufgeführt, muss sie auf dem Gerät verfügbar sein.

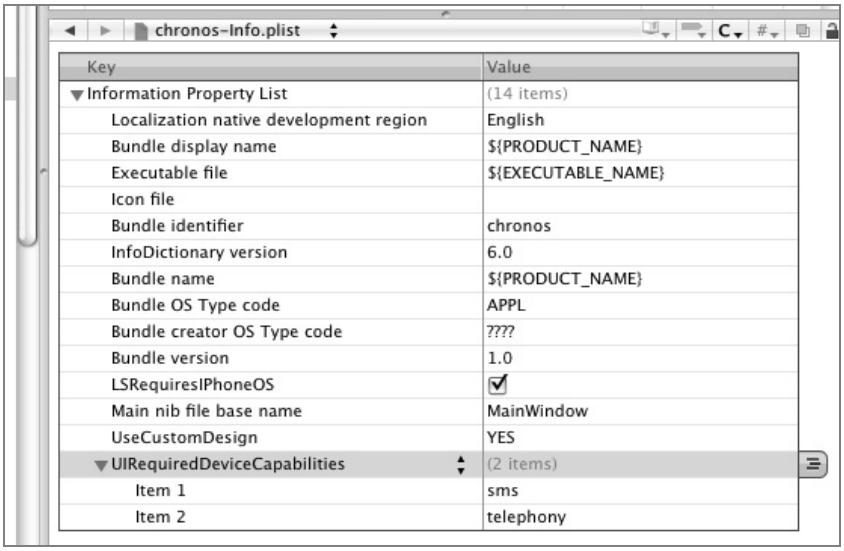


Bild 25.1: Notwendige Gerätefunktionen enthält der Key `UIRequiredDeviceCapabilities`

Die folgenden Einträge sind möglich:

- `telephony`
- `sms`
- `still-camera`
- `autofocus-camera`
- `video-camera`
- `wifi`
- `accelerometer`
- `location-services`
- `gps`
- `magnetometer`
- `microphone`
- `opengles-1`
- `opengles-2`
- `armv6`
- `armv7`
- `peer-peer`

Die Schlüssel sollten nur für die Kernfunktionalität der App verwendet werden. Falls eine App trotz einer fehlenden Hardwarekomponente sinnvoll nutzbar ist, ist es besser, lediglich die Ausführung des betroffenen Codes zu verhindern.

25.1.2 Log-Meldungen

Vor dem Build sollten alle Log-Meldungen, die noch zu Debug-Zwecken im Code vorhanden sind, entfernt oder zumindest auskommentiert werden. Unnötige Log-Meldungen kosten Performance und Speicherplatz.

25.1.3 Unit Tests

Sollten Unit Tests für die App existieren, ist nun ein guter Moment, sie nochmals laufen zu lassen.

25.1.4 Versionsverwaltung

Der letzte Stand aller Dateien wird in die Versionsverwaltung eingecheckt. Anschließend wird dieser Stand mit einem *Tag* versehen. Auf diese Weise wird sichergestellt, dass der Code, der zur Version im App Store geführt hat, auch nach späteren Änderungen jederzeit wieder identifiziert werden kann. Wie schon im Kapitel »Beta-Test« besprochen, wird zum Erzeugen eines Tags in Subversion der aktuelle Stand unter *trunk* einfach unter einem neuen Namen in das Verzeichnis *tags* kopiert:

```
svn copy trunk/ tags/NamesDesTags
```

25.2 App-Store-Distributionsprofil

Für die Verteilung über den App Store kann nicht das Ad-hoc- oder Entwicklungsprofil verwendet werden. Es wird ein *Provisioning Profile* für die App Store Distribution benötigt.

Dieses wird wie das Ad-hoc-Profil im Provisioning-Bereich des *Developer Program Portal* unter dem Reiter *Distribution* angelegt. Als Distributionsmethode wird diesmal allerdings *App Store* ausgewählt:

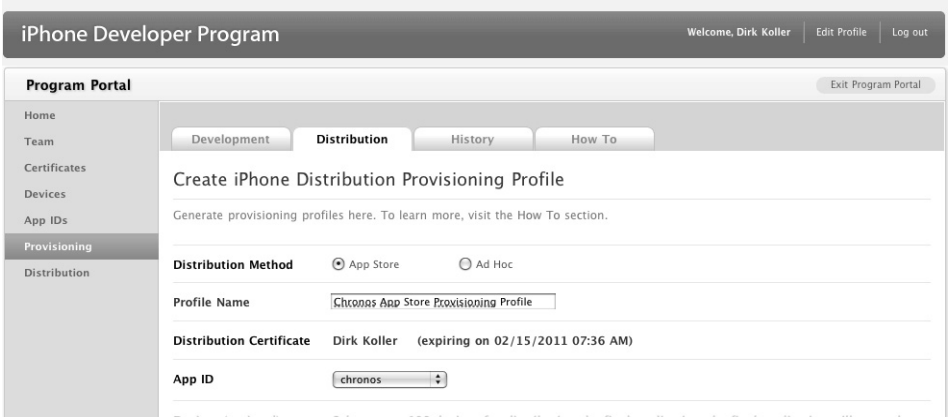


Bild 25.2: Das App Store Distribution Provisioning Profile

Das fertige Profil wird, wie schon beschrieben, von der Seite heruntergeladen und in Xcode installiert (also per Drag & Drop zum Beispiel in den Organizer unter *Provisioning Profiles* gezogen). Kontrollieren Sie wie immer, ob es dort auch angekommen ist.

25.3 Der finale Build

Für den App Store Build wird am besten eine weitere Kopie der Release-Konfiguration, zum Beispiel mit Namen *App Store Distribution*, gemacht.

Für die neue Konfiguration muss das neue Profil wieder in den Projektinformationen unter *Code Signing Identity* ausgewählt werden. Auch dieser Schritt wurde schon mehrfach beschrieben. Achten Sie beim Setzen des Profils darauf, dass Sie auch wirklich die *App Store Distribution*-Konfiguration in der Auswahlliste links oben im Projekt-Info-Dialog ausgewählt haben. Ansonsten verändern Sie die Einstellungen der anderen Konfigurationen:

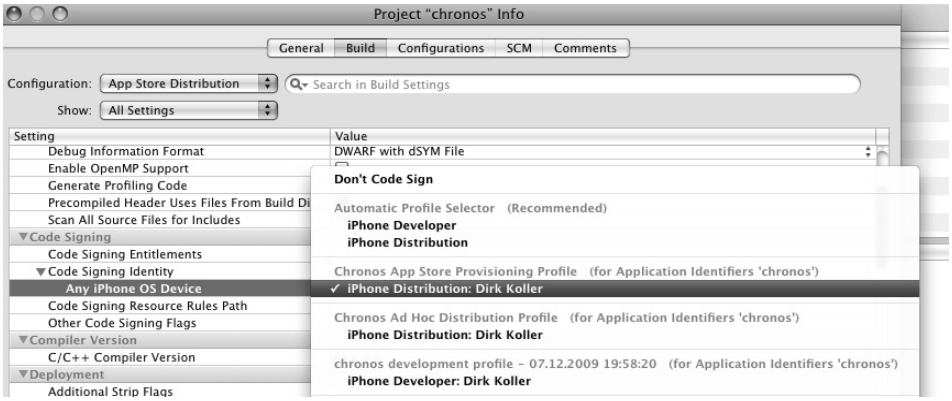


Bild 25.3: Code Signing Identity für den App Store Build

Die neue Konfiguration muss außerdem als aktive Konfiguration in der *Overview*-Auswahlliste in der Toolbar von Xcode ausgewählt werden. Als *Active SDK* wird an gleicher Stelle das gewünschte Gerät ausgewählt. Falls Sie zuvor Unit Tests ausgeführt haben, muss möglicherweise das Target wieder zurückgesetzt werden.

Nach einem letzten Clean darf dann die *Build*-Schaltfläche betätigt werden.

Mit Rechtsklick auf das erzeugte Produkt (unter *Products* im Projektordner in *Groups & Files*) kann das Binary über die Funktion *Reveal in Finder* lokalisiert werden.

25.4 Vor dem Einstellen

Neben dem eigentlichen Binary wird für das Einstellen der App in den App Store eine ganze Reihe weiterer Informationen benötigt. Es ist sinnvoll, sich vorher Gedanken über die abgefragten Daten zu machen und diese dann einfach per Copy & Paste aus einer Textdatei in die Webseite hineinzukopieren. Ansonsten ist der Ärger groß, wenn man nach ein paar Minuten hektischen Überlegens vom Server wegen eines Timeouts hinausgeworfen wird. Neben den Texten werden auch einige Grafiken benötigt, die ebenfalls bereitliegen sollten.

Vielleicht nehmen Sie sich vor dem Erfassen kurz Zeit, um die Seite einer beliebigen App im App Store näher zu untersuchen. Dadurch bekommen Sie eine gute Vorstellung davon, wie Ihre Daten präsentiert und verwertet werden.

Im Einzelnen sollten die folgenden Informationen, gegebenenfalls in mehreren Sprachen, zur Verfügung stehen:

Application Name

Der Name der App.

Application Description

Die Beschreibung der Anwendung. Sie kann maximal 4.000 Zeichen lang sein.

Primary & Secondary Category

Die Apps im App Store sind in Kategorien unterteilt. An dieser Stelle können zwei möglichst treffende Kategorien ausgewählt werden. Die zweite Kategorie ist optional. Zur Verfügung stehen:

- *Book*
- *Business*
- *Education*
- *Entertainment*
- *Finance*
- *Games*

- *Healthcare & Fitness*
- *Lifestyle*
- *Medical*
- *Music*
- *Navigation*
- *News*
- *Photography*
- *Reference*
- *Social Networking*
- *Sports*
- *Travel*
- *Utilities*
- *Weather*

Für *Chronos* bieten sich beispielsweise *Utilities* und *Business* an.

Copyright

Die Copyright-Informationen für die App. Es handelt sich dabei um den Namen der Person oder des Unternehmens, die bzw. das die Rechte an der App hat. Das Datum wird vorangestellt.

Beispiel: *2010 Dirk Koller*

Version Number

Die Versionsnummer der App, wie sie im App Store angezeigt wird.

Beispiel: *1.0*

SKU Number

Eine Bezeichnung, anhand der mehrere eigene Apps im App Store eindeutig auseinandergehalten werden können. Die SKU-Nummer ist eine interne Information, die der Benutzer der App nicht sieht. Sie taucht zum Beispiel in den Finanzreports auf. Bei der Eingabe sind übrigens auch alphanumerische Zeichen erlaubt. Außerdem ist wissenswert, dass die *SKU Number*, einmal vergeben, nicht mehr geändert werden kann.

Beispiel: *CHRONOS_1*

Keywords

Bei der unglaublichen Anzahl von Apps das Gewünschte zu finden, ist gar nicht so einfach. Apple hat deswegen nachträglich Keywords eingeführt, die dem Benutzer bei der Suche nach einer App helfen.

Für Chronos könnten das sein: *Zeiterfassung, Arbeitszeit*

Application URL

Die *Application URL* ist die Adresse der Homepage der App. Die Angabe der URL ist optional. Hier kann die App zu Werbezwecken etwas ausführlicher vorgestellt werden. Für eine schnelle erste Präsentation können Sie die Beschreibung und Screenshots auf der Webseite anzeigen.

Support URL

Die *Support URL* gibt eine Seite an, auf der der Benutzer Hilfe zur App erhält. Das könnte so etwas wie eine kleine Bedienungsanleitung, zumindest für die versteckteren Funktionen oder eine Liste häufig gestellter Fragen sein. Vielleicht hatten die Beta-Tester mit der einen oder anderen Funktion Probleme? Nutzen Sie die Gelegenheit, potenzielle Fragen im Voraus zu beantworten.

Support Email Address

Eine E-Mail-Adresse, unter der Apple Sie erreichen kann, wenn Fragen zur oder Probleme mit der App auftreten. Die Adresse ist für Benutzer nicht sichtbar.

Demo Account – Full Access

Apple prüft bekanntermaßen alle Apps vor der Aufnahme in den App Store. Abhängig von der Art der Anwendung werden zur Prüfung möglicherweise Zutrittsinformationen wie Benutzername oder Passwörter benötigt. Diese Informationen können dem Reviewer hier mitgeteilt werden.

Large 512 x 512 Icon

Hier handelt es sich um eine 512 x 512 Pixel große Version des Anwendungs-Icons. Die Auflösung muss mindestens 72 dpi betragen und die Grafik vom Typ JPEG oder TIFF sein. Das Icon wird im iTunes App Store angezeigt. Gerüchten zufolge wurden schon Apps abgelehnt, bei denen das Aussehen des großen Icons drastisch von dem des normalen App-Icons abwich. Ein wenig optisch ansprechender darf das große Icon aber schon sein.

Screenshots

Insgesamt können bis zu fünf Screenshots für die App hochgeladen werden. Von der Möglichkeit sollte auch unbedingt Gebrauch gemacht werden, weil es eines der wichtigsten Kriterien bei der Entscheidung des Benutzers für oder gegen eine App ist.

Die Screenshots müssen vom Typ `.jpeg`, `.jpg`, `.tif`, `.tiff`, oder `.png` sein und 320 x 480, 480 x 320, 320 x 460 oder 480 x 300 Pixel groß sein, ebenfalls mit mindestens 72 dpi.

Wie die Screenshots angefertigt werden können, wurde im Kapitel »Icons, Farben & Schriften« bereits besprochen. Für die Präsentation im App Store sollte die Status Bar entfernt werden.

Pricing / Availability

Zu guter Letzt sollten Sie sich Gedanken machen, was Ihr Programm kosten soll (falls es sich nicht um ein kostenloses Angebot handelt). Die Preise sind in Preisklassen (*Tiers*) unterteilt. Bis Preisklasse 50 entspricht die Tier-Nummer dem Dollarpreis (eine Tier 34 App kostet $33,99). Ab Tier 50 steigen die Preise bis zu Tier 85 ($999,99) steiler an. Eine genaue Übersicht über die Preise in den verschiedenen Preisklassen und Währungen bietet die *App Store Pricing Matrix* in *iTunes Connect*.

25.5 Einstellen

Liegen alle Daten vor, geht es ans eigentliche Einstellen. Im *iPhone Dev Center* findet sich der Link *iTunes Connect* [URL-CONNECT]. *iTunes Connect* ist das Portal zum Verwalten der eigenen Apps im App Store. Von hier aus können zum Beispiel Verkaufs- oder Downloadzahlen eingesehen, Promotioncodes angefordert oder neue Apps eingestellt werden.

Falls Sie planen, kostenpflichtige Apps zu vertreiben, ist es eine gute Idee, so früh wie möglich das Modul *Contracts, Tax & Banking Information* aufzusuchen. Die dort von Ihnen zu hinterlegenden Informationen benötigen einige Zeit zur Bearbeitung. Erst wenn die Bearbeitung abgeschlossen ist, kann die App vertrieben werden. Für eine kostenlose Anwendung braucht das Modul nicht beachtet zu werden.

Hier interessiert im Moment die Funktionalität zum Einstellen neuer Apps. Das entsprechende Modul heißt *Manage Your Applications*.

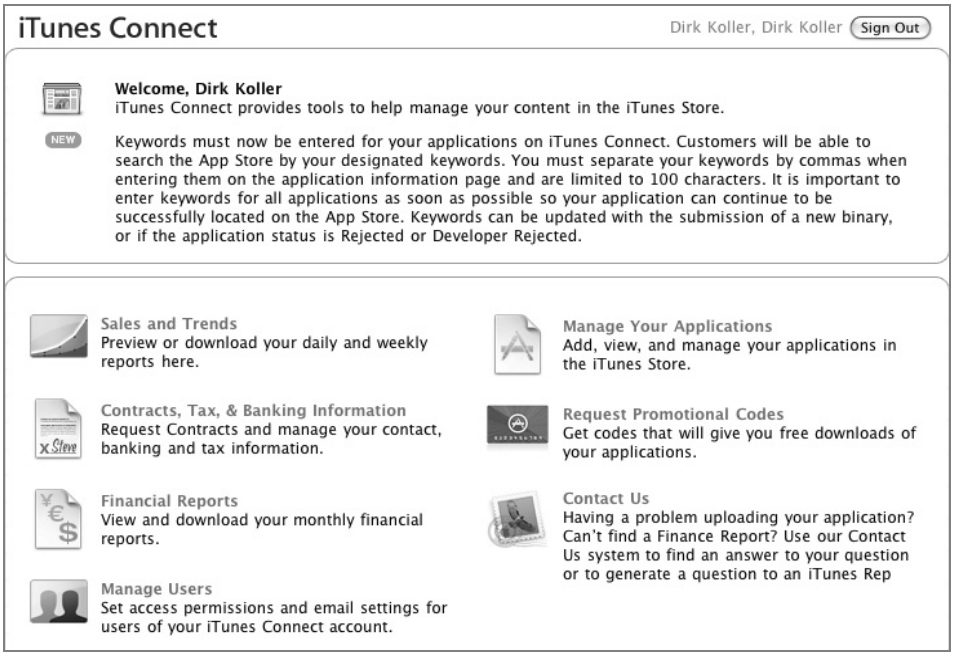


Bild 25.4: iTunes Connect

Im *Manage Your Application*-Bereich werden die bereits eingestellten Apps aufgelistet. Bitte beachten Sie die Downloadmöglichkeit für den *User Guide* am unteren Ende der Seite. Sollten die hier aufgeführten Informationen nicht ausreichen, finden Sie dort weitere Erklärungen und Beispiele.

Der Button *Add New Application* erlaubt das Zufügen von weiteren Programmen. Wird er betätigt, wird man beim Einstellen der ersten App (und nur dann!) nach der bevorzugten Sprache und dem Firmennamen gefragt. Die Frage nach dem Firmennamen ist etwas tückisch, weil dieser im Moment nicht über iTunes Connect geändert werden kann. Eventuell kann die Hotline bei Änderungswünschen weiterhelfen. Am besten aber machen Sie sich vorher darüber Gedanken, unter welchem Namen (zum Beispiel als Entwickler oder als Firma) Sie Ihre Produkte im App Store vertreiben möchten.

Als Nächstes wird man mit den Exportbeschränkungen konfrontiert:

Bild 25.5: Exportbestimmungen

Wenn die App keine Kryptografie enthält, kann die Frage mit *Nein* beantwortet und mit *Continue* zur Seite *Overview* gewechselt werden:

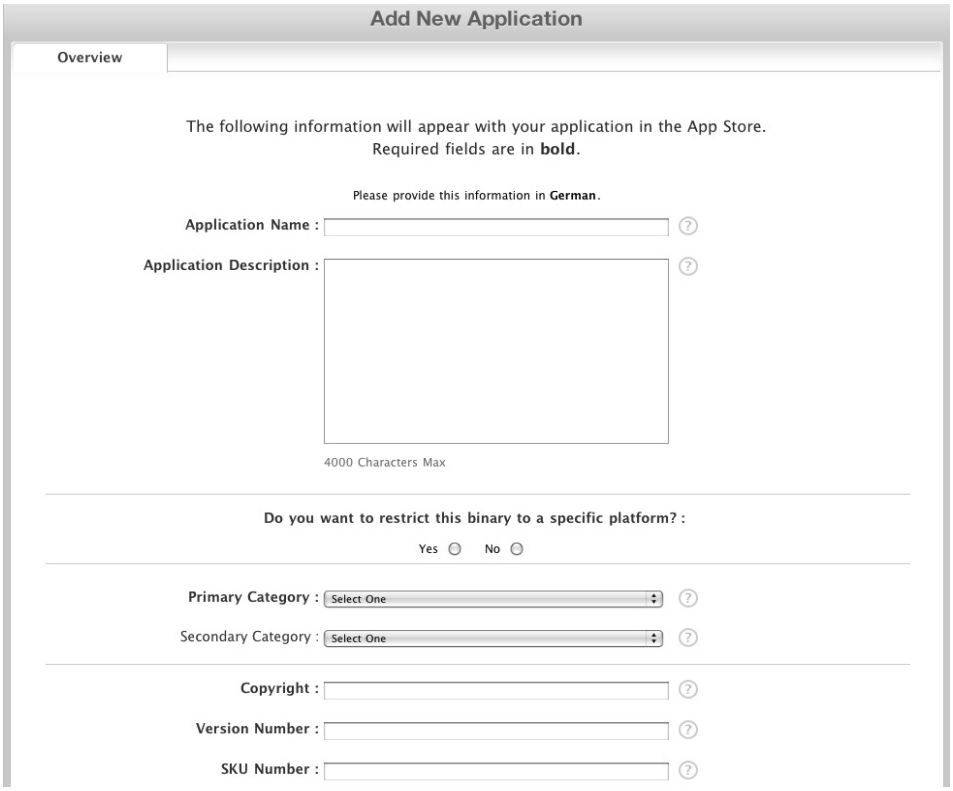


Bild 25.6: Overview-Seite

Hier werden nun die zuvor gesammelten Daten in die entsprechenden Textfelder kopiert bzw. in den Listenfeldern ausgewählt. Die Seite *Overview* bietet außerdem die Möglichkeit, eigene Lizenzbestimmungen einzugeben. Wird darauf verzichtet, gilt das Standard-EULA (End User License Agreement). Es kann unter [URL-EULA] eingesehen werden.

Danach folgt die Seite *Ratings* mit Fragen zu Gewalt, Sex, Drogen oder Horror in der App. Daraus ergibt sich die im App Store angezeigte Altersfreigabe.

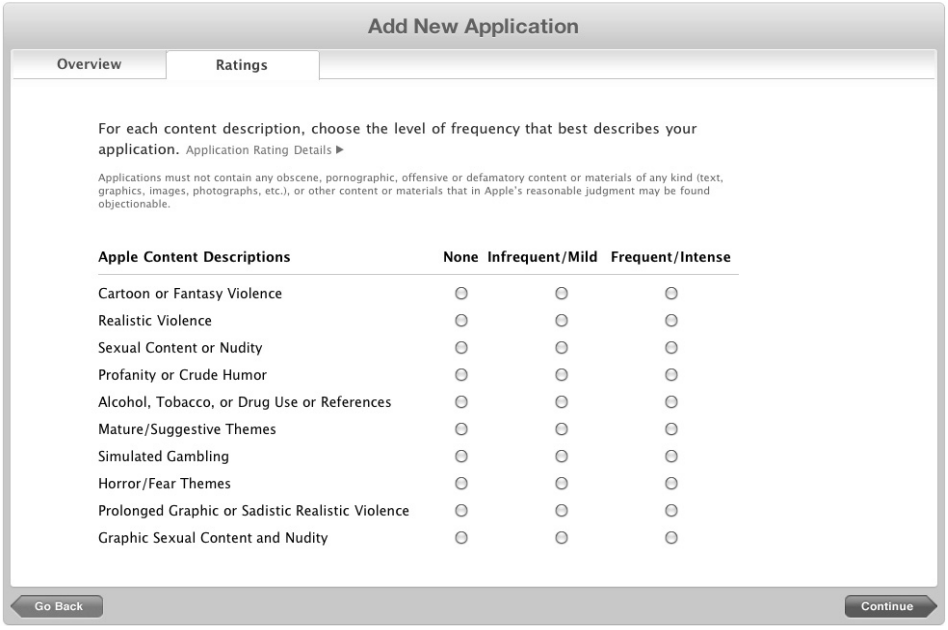


Bild 25.7: Ratings-Seite

Die darauf folgende Seite heißt *Upload*. Hier werden die beim Build erzeugte Binärdatei und die verschiedenen Grafiken (Large Icon & Screenshots) hochgeladen. Das *Large Icon* und der *Primary Screenshot* sind Pflicht. Alle anderen Dateien (auch das Binary!) können später hinzugefügt werden.

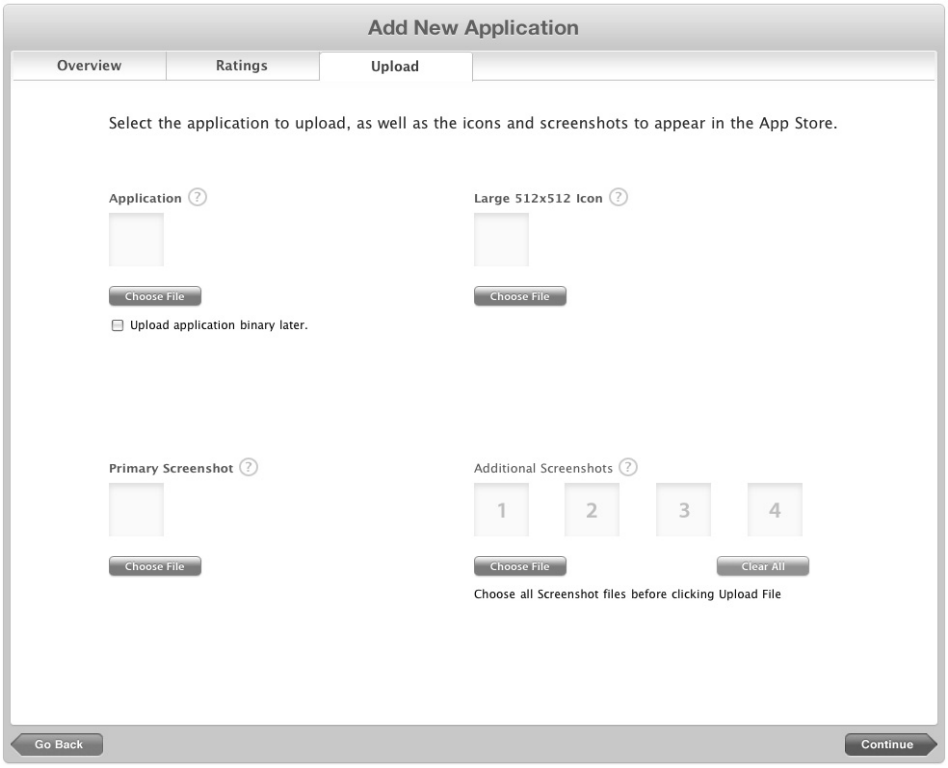


Bild 25.8: Upload-Seite

Nun folgt die Seite *Pricing*, die zur Aufnahme des Verfügbarkeitsdatums und des Preises dient.

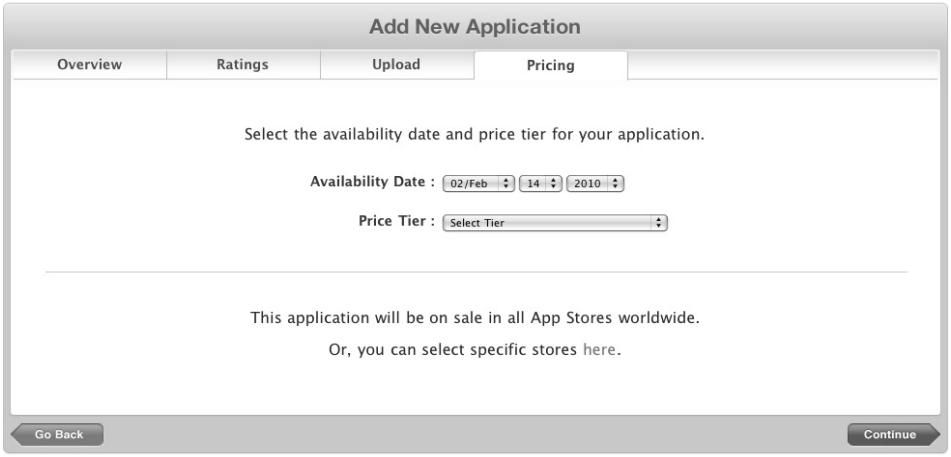


Bild 25.9: Pricing-Seite

Die letzte Seite mit Namen *Localization* bietet die Möglichkeit, die eingegebenen Daten in weiteren Sprachen zur Verfügung zu stellen.

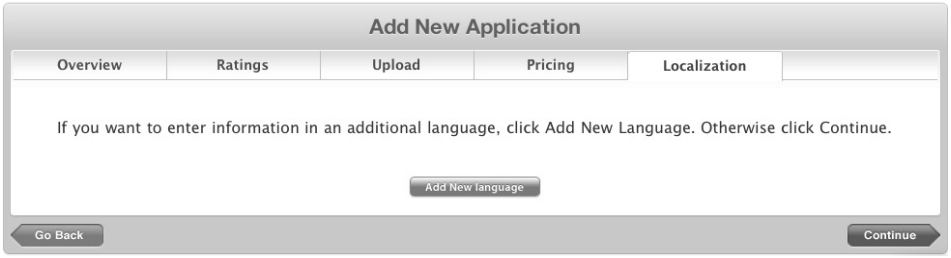


Bild 25.10: Localization-Seite

25.6 Der Approval-Prozess

Nach dem Einstellen bleibt nicht mehr zu tun, als zu warten. Vielleicht nutzen Sie auch die Zeit, um die Webseite für die App anzulegen. Apple wird irgendwann in den nächsten (hoffentlich wenigen) Tagen oder Wochen Ihre Anwendung testen. Wie lange es genau dauert, bis die Anwendung geprüft wird, kann nicht genau gesagt werden. Gerüchten zufolge kann das nach einiger Kritik am Prozess inzwischen auch innerhalb von zwei Tagen geschehen. Nachfragen bei Apple beschleunigen den Prozess wohl eher nicht.

Im Test wird überprüft, ob Sie sich an die Spielregeln von Apple gehalten haben. Die App darf keine privaten Funktionen benutzen, keine pornografischen Inhalte verbreiten und vieles mehr. All das wird geregelt im *iPhone Developer Program License Agreement* (iDPLA).

Auch Verstöße gegen die *Human Interface Guidelines* werden geahndet. Wenn Ihre App beispielsweise auf Daten im Internet zugreift und dieses nicht verfügbar ist, muss der Benutzer mit einer Fehlermeldung darauf hingewiesen werden. Geschieht das nicht, liegt ein Mangel vor, der den Eintritt in den App Store verhindert. Frust über eine App kann von Frust über das Gerät nicht sauber getrennt werden. Deshalb setzt Apple hier im eigenen Interesse wenigstens ein paar grundlegende Regeln durch.

Hat der Reviewer Einwände, können Sie mit einer E-Mail rechnen, die das Problem beschreibt. Es testen also wirklich echte Menschen die Anwendung und nicht (nur) Rechner.

Das Schlimme an einer Ablehnung ist die erneute Wartezeit vor dem nächsten Test. Auch wenn der eigentliche Mangel in wenigen Minuten behoben sein kann, muss womöglich wieder 14 Tage gewartet werden, bis der nächste Test erfolgt.

Wurde die App in den Store aufgenommen, werden Sie ebenfalls mit einer E-Mail benachrichtigt. Sie sollten dann die App aus dem App Store auf Ihr Gerät laden, um zu schauen, ob alles wie gewünscht klappt. Notfalls muss der Verteilungsprozess gestoppt und das Problem behoben werden.

26 Besonderheiten bei der iPad-Entwicklung

Im Januar 2010 hat Apple ein weiteres Gerät mit iPhone OS als Betriebssystem vorgestellt: den lange erwarteten Tablet-PC namens iPad. Wie nach innovativen Neuvorstellungen üblich, ist in der Blogosphäre erst mal ein Riesenstreit darüber entbrannt, ob ein solches Gerät, das zwischen Smartphone und Notebook angesiedelt ist, überhaupt gebraucht wird. Letztlich werden die Käufer darüber entscheiden, und es darf spekuliert werden, ob es sich dabei wirklich um die technikverliebten Blogger oder aber vielleicht um eine ganz andere Zielgruppe handelt: All jene Durchschnittsbürger, die einfach auf der Couch ihre Mails checken oder ein wenig im Internet surfen und sich dabei nicht in den Tiefen eines komplett offenen Systems verirren möchten. Gut möglich also, dass das »magische Gerät« (O-Ton Apple-Chef Steve Jobs bei der Vorstellung) erneut ein Riesenerfolg werden wird.

Wie auch immer: In diesem Kapitel sollen die (wenigen) Unterschiede in der Entwicklung gegenüber iPhone und iPod Touch besprochen werden. Neben dem Anlegen von iPad-Projekten und dem Portieren von bestehenden Apps stehen insbesondere die neuen UI-Elemente wie *Popovers* oder *Split Views* im Fokus. Andere Themen wie die erweiterte Gestenerkennung oder neue Grafik-Features werden gemäß der Ausrichtung dieses Buchs außen vor gelassen oder nur kurz angesprochen.

26.1 Erstellen eines iPad-Projekts

Für viele Anwendungszwecke ist die geringe Auflösung von iPhone und iPod touch einfach nicht geeignet. Oder möchten Sie eine Textverarbeitung auf einem Smartphone bedienen? Anders ist die Situation auf dem iPad. Das größere Display erlaubt völlig neue Anwendungen, die nur auf dem neuen Tablett sinnvoll sind.

Um ein »reinrassiges« iPad-Projekt anzulegen, steht seit dem SDK 3.2 die neue Projektvorlage *Split view based Application* zur Verfügung. Außerdem erlauben seit diesem Release einige der altbekannten Vorlagen die Wahl zwischen iPad oder iPhone als Zielprodukt. Die folgende Abbildung zeigt diese Wahlmöglichkeit beim Erstellen einer *Window-based Application* mit dem New Project-Assistenten:

Bild 26.1: Anlegen einer Window-basierten Applikation für das iPad

Nach dem Erstellen des Projekts mit Hilfe der Vorlage läuft die weitere Entwicklung wie gewohnt ab. Die gleichen Dateien werden vom Assistenten erzeugt, die im Kapitel 8 beim Erzeugen des iPhone-Projekts mit dieser Vorlage besprochen wurden. Bei der Arbeit mit dem Interface Builder fallen in erster Linie die neuen Dimensionen von View und Window auf. Das neue Projekt lässt sich sofort auf dem Simulator ausprobieren. Dieser enthält im Menü *Hardware* ab SDK 3.2 den Menüpunkt *Gerät*, unter dem *iPad* oder *iPhone* auswählbar sind.

Bild 26.2:
Der iPad-Simulator

26.2 Portierung von iPhone-Apps

Angesichts der erweiterten technischen Möglichkeiten (und natürlich auch der kommerziellen Chancen) des iPads werden wohl viele Entwickler über eine Portierung vorhandener iPhone-Apps auf das Tablett nachdenken.

Zwar laufen bestehende Anwendungen im Kompatibilitätsmodus auch ohne Portierung auf dem iPad, allerdings ist das eher als Notlösung zu betrachten. Die Apps werden hierbei entweder in der Mitte des iPads als kleines Fenster mit 480 x 320 Pixeln dargestellt oder aber vergrößert, um den Platz zu füllen. Die erste Variante führt zu einem unschönen dicken Trauerrahmen um die Anwendung. Nach Betätigen des 2x-Buttons wird der Platz zwar genutzt, aber die Komponenten werden in unangemessener Größe dargestellt.

Letztlich müssen also bei der Portierung einer bestehenden App die Views überarbeitet werden. Zum Glück hat Apple hier schon ein wenig Vorarbeit geleistet. Wir nutzen im Folgenden das Chronos-Projekt, um uns die Portierung anzusehen. Nach dem Öffnen der Anwendung in Xcode klickt man das Target der App mit der rechten Maustaste an und wählt die Option *Upgrade Current Target for iPad...* aus:

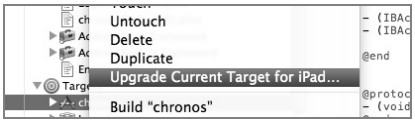


Bild 26.3: Die Option *Upgrade Current Target for iPad*

Der daraufhin folgende Dialog stellt den Entwickler vor eine schwierige Wahl – ob nämlich aus dem vorhandenen Projekt eine *Universal Application* (ein Binär-File, das unverändert auf iPhone und iPad läuft) werden soll oder aber ob ein weiteres Target für das iPad zugefügt werden soll, so dass zwei separate Binärfiles daraus resultieren:

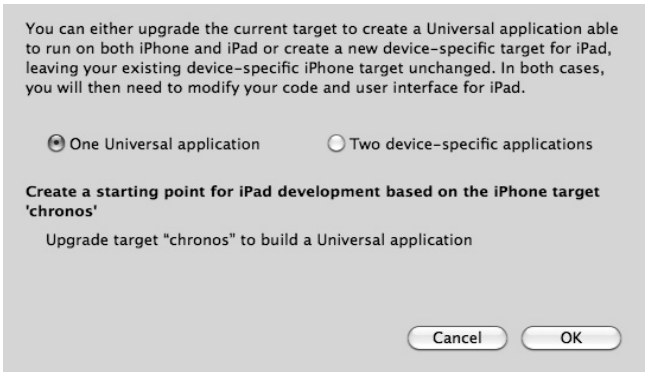


Bild 26.4: Upgrade-Möglichkeiten

Wofür Sie sich auch entscheiden, in jedem Fall kommt einiges an Arbeit auf Sie zu.

Das Universal Binary ist für den Benutzer einfacher zu handhaben, weil er nur eine Anwendung für alle Geräte laden muss. Der Entwicklungsaufwand ist allerdings ein wenig höher. Der Upgrade-Prozess übernimmt das Setzen einiger Build-Properties (Targeted Device Family, Base SDK) sowie das Erzeugen eines `Main`-Nib-Files für das iPad (`MainWindow-iPad.xib` in der Gruppe *Resources- iPad*). Dieses wird beim Start des Projekts auf dem iPad benutzt und erwartungsgemäß auch in `chronos-Info.plist` aufgelistet. Weitere Nib-Files müssen aber vom Entwickler selber angelegt und an die Gegebenheiten des neuen Geräts angepasst werden Außerdem muss in den View Controllern natürlich das Nib-File für das richtige Gerät geladen werden.

Um zu verhindern, dass Code, der nur auf dem iPad funktioniert, auf dem iPhone ausgeführt wird, muss an allen betreffenden Stellen über Runtime Checks überprüft werden, auf welchem Gerät der Code aktuell läuft. Mit Hilfe der Funktion `NSClassFromString` kann überprüft werden, ob eine Klasse zur Laufzeit vorhanden ist:

```
Class splitVCClass = NSClassFromString(@"UISplitViewController");
if (splitVC)
{
   UISplitViewController* splitViewController = [[splitVCClass alloc] init];
   …
}
```

Ähnlich kann mit Hilfe der Methode `instancesRespondToSelector:` geprüft werden, ob bei bereits vor dem SDK 3.2 existierenden Klassen eine neu eingeführte Methode zur Verfügung steht.

Entscheidet man sich für zwei Targets, dann müssen einzelne Klassen (beispielsweise View Controller) gedoppelt und den unterschiedlichen Targets zugeordnet werden. Durch bedingte Kompilierung kann abhängig vom Base SDK erreicht werden, dass entweder der richtige View Controller erzeugt wird oder aber bei Verwendung des gleichen View Controllers das richtige Nib-File geladen wird:

```
#if __IPHONE_OS_VERSION_MAX_ALLOWED >= 30200
// iPad View Controller erzeugen oder iPad-Nib-File laden
#else
// iPhone View Controller erzeugen oder iPhone -Nib-File laden
#endif
```

Der Upgrade-Prozess im Falle zweier separater Anwendungen legt iPad-Varianten aller Nib-Files an. Diese berücksichtigen aber nur die höhere Auflösung und sind in jedem Fall zu überarbeiten. Außer dem Code und den Nib-Files müssen auch die Grafiken an die neue Display-Größe angepasst werden: Application Icons müssen auf dem iPad 72x72 Pixel groß sein.

Neben den beiden beschriebenen Möglichkeiten (Universal Application oder zwei Targets in einem Projekt) ist es natürlich auch denkbar, den Code für die iPhone- und die iPad-App in zwei separaten Xcode-Projekten vorzuhalten. Dabei sollte man natürlich den enormen Aufwand zur Pflege der doppelten Klassen und Ressourcen bedenken.

Nach dem Upgrade kann das Projekt schon auf dem Simulator gestartet werden. Die folgende Abbildung zeigt das Beispielprojekt auf dem Simulator in der iPad-Variante:

Bild 26.5: Die Anwendung *Chronos* im iPad-Simulator

Der Trauerrahmen ist weg und das User Interface wird in vernünftiger Größe dargestellt. Trotzdem macht schon der erste Eindruck klar, dass es wohl mit dem einfachen Upgrade nicht getan ist. Die Menüpunkte wirken in der linken oberen Ecke verloren. Das Display des iPads hat eine Auflösung von 1024 x 768 Pixeln. Das vergrößerte Platzangebot des Geräts macht deshalb ein komplettes Überdenken des User Interfaces notwendig.

Abgesehen von dem unbefriedigenden Erscheinungsbild funktioniert die App aber wie gewohnt.

26.3 Erweiterungen für das iPad

Glücklicherweise bringt das mit dem iPad vorgestellte iPhone SDK 3.2 keine großen Änderungen, sondern vorwiegend Erweiterungen für das iPad mit. Alle Klassen, die für die iPhone-Entwicklung existieren, können auch auf dem iPad genutzt werden. Umgekehrt gilt das nicht, die im SDK 3.2 neu vorgestellten Klassen funktionieren nicht auf iPhone oder iPod touch. Anwendungen, die für das iPhone geschrieben wurden, laufen somit fast immer auch auf dem iPad und können optional an die Gegebenheiten des neuen Geräts angepasst werden.

26.3.1 Split Views

Split Views erlauben die Aufteilung des Screens in zwei Bereiche. Somit hat man zum Beispiel die Möglichkeit, Master-Detail-Beziehungen gleichzeitig auf dem Screen darzustellen. Der linke Bereich ist 320 Pixel breit, der rechte füllt den Rest. Wie beim *Navigation Controller* und beim *Tab Bar Controller* handelt es sich bei diesem neuen Typ um einen Container-Controller, der andere *View Controller* aufnimmt und deren Views präsentiert.

Zur Demonstration des Controllers wird im Folgenden das Chronos-Hauptmenü als erster View und eine Liste als zweiter View angezeigt. Das ist für eine Chronos-iPad-Adaption sicherlich nicht der Weisheit letzter Schluss, soll aber zur Besprechung des *Split View Controllers* genügen.

Um ein Split View-UI zu erzeugen, zieht man am einfachsten einen *Split View-Controller* aus der Interface Builder-Library in das Document Window. Im Fall der Demoanwendung handelt es sich um das Document Window des beim Upgrade erzeugten Nib-Files `MainWindow-iPad.xib`.

Bild 26.6: Split View Controller in der Library

Um den neuen View dem Haupt-Window der App zufügen zu können, müssen ein Outlet vom Typ `UISplitViewController` (in `chronosAppDelegate`) und die entsprechende Connection angelegt werden:

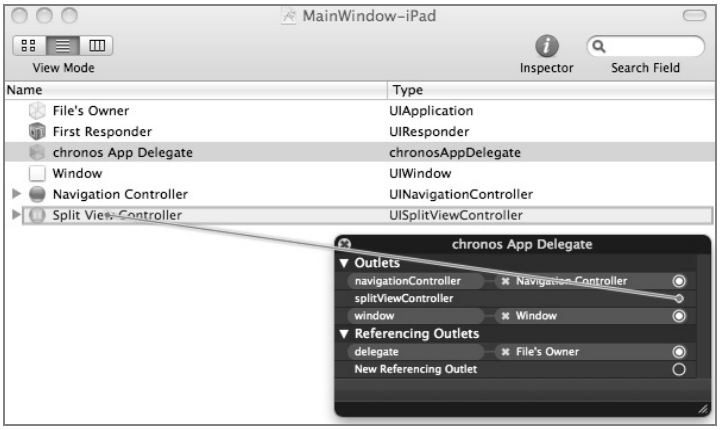


Bild 26.7: Connection zum Split View Controller

Anschließend kann der View dem Window in gewohnter Weise als Subview zugefügt werden:

```
[window addSubview:[splitViewController view]];
```

Der *Split View Controller* aus der Interface Builder-Library enthält bereits zwei Default-*View Controller*. Der Typ dieser Controller wird wie gehabt über die *Class Identity*-Property auf die gewünschten Klassen (hier `MenuTableViewController` und `KundenTableViewController`) des Projekts gesetzt. Die folgende Abbildung zeigt das Split View-UI in der Landscape-Orientierung:

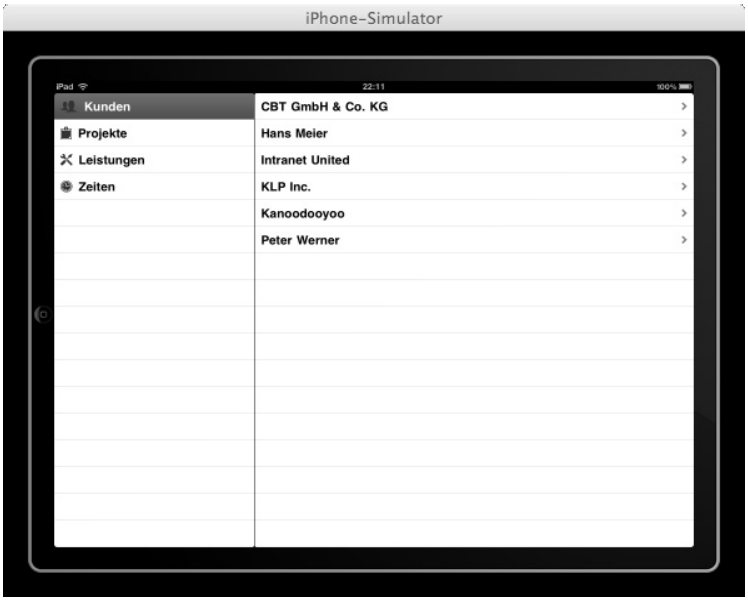


Bild 26.8: Split View

In der Portrait-Orientierung wird aus Platzgründen nur der rechte View angezeigt. Das Ausblenden des linken Bereichs geschieht automatisch, der Entwickler muss sich darum nicht kümmern. Möchte man dem User in dieser Orientierung trotzdem den linken View anzeigen, ist dies zum Beispiel mit den neuen Popovers möglich. Das Protokoll `UISplitViewControllerDelegate` ermöglicht es, auf das Verschwinden und Anzeigen des Bereichs zu reagieren. Die folgenden (optionalen) Methoden sind im Protokoll enthalten:

- `splitViewController:willHideViewController:withBarButtonItem:forPopoverController:`
- `splitViewController:willShowViewController:invalidatingBarButtonItem:`
- `splitViewController:popoverController:willPresentViewController:`

Um den View in einem Popover zu präsentieren, kann man die erste der drei Methoden implementieren und das als Parameter übergebene *Bar Button Item* einer Tool Bar zufügen. Das neue Item in der Tool Bar ist dann für das Anzeigen des Popover mit dem linken View zuständig. Die Methode wird aufgerufen, wenn der linke View ausgeblendet wird, also beim Wechsel von der Landscape- in die Portrait-Orientierung:

```
- (void)splitViewController: (UISplitViewController*)svc
   willHideViewController:(UIViewController *)aViewController
   withBarButtonItem:(UIBarButtonItem*)barButtonItem forPopoverController:
   (UIPopoverController*)pc {

    barButtonItem.title = @"Root List";
    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items insertObject:barButtonItem atIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = pc;
}
```

In der zweitgenannten Methode wird das *Bar Button Item* beim Wechsel in die Landscape-Orientierung wieder aus der Tool Bar entfernt:

```
- (void)splitViewController: (UISplitViewController*)svc
   willShowViewController:(UIViewController *)aViewController
   invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem {

    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items removeObjectAtIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = nil;
}
```

Ein komplettes Beispiel für diese Vorgehensweise findet sich in der Projektvorlage *Split View-based Application*.

Damit der *Split View* auf das Drehen des Geräts reagiert, müssen beide im *Split View Controller* enthaltenen *View Controller* die Methode `shouldAutorotateToInterfaceOrientation:` überschreiben und `YES` zurückliefern.

26.3.2 Popovers

Popovers sind kleine Fenster, die zeitweise über andere Views gelegt werden. Sie können zum Beispiel Werkzeugpaletten oder Konfigurationsoptionen zur Verfügung stellen. Aber auch die Darstellung des nicht angezeigten Teils eines Split Views in der Portrait-Orientierung gehört, wie besprochen, zu den typischen Aufgaben des Popovers. Im Inhaltsbereich des Popovers werden andere View Controller (bzw. die zugehörigen Views) angezeigt. Dabei kann es sich um eigene View Controller, Navigation Controller, Tab Bar Controller oder Table View Controller handeln.

Die folgende Abbildung zeigt den in Kapitel 20 erstellten View Controller zur Verwaltung von In-App-Settings in einem Popover:

Bild 26.9: Settings im Popover

Der zugehörige Code sieht folgendermaßen aus:

```
UIViewController *controller = [[InAppSettingsViewController alloc]
   initWithNibName:@"InAppSettingsView" bundle:nil];
UIPopoverController* aPopover = [[UIPopoverController alloc]
   initWithContentViewController:controller];
aPopover.delegate = self;
[controller release];
[aPopover presentPopoverFromBarButtonItem:settingsItem
   permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
```

Zunächst wird der View Controller für die Settings erzeugt. Der anschließend erzeugte `UIPopoverController` bekommt in der Methode `initWithContentViewController:` den anzuzeigenden View Controller übergeben. Danach bekommt der Popover Controller die eigene Klasse als Delegate gesetzt und wird mit der Methode `presentPopoverFromBarButtonItem:permittedArrowDirections:animated:` zur Anzeige gebracht. Man beachte hierbei die übergebene Instanz des `UIBarButtonItems` namens `settingsItem`. Anhand dieses Parameters »weiß« das Popover, durch welchen Button es ausgelöst wurde. Diese Verbindung wird durch einen kleinen Pfeil symbolisiert. Um das Popover zum Verschwinden zu bringen, bedarf es lediglich eines Taps neben das Popover.

Die als Delegate bestimmte Klasse muss das Protokoll `UIPopoverControllerDelegate` mit den optionalen Methoden `popoverControllerShouldDismissPopover:` und `popoverControllerDidDismissPopover:` implementieren. Sie werden aufgerufen, bevor bzw. nachdem das Popover ausgeblendet wird. Wird in ersterer Methode ein `NO` zurückgegeben, wird dadurch das Ausblenden des Popovers verhindert.

Während das Popover sichtbar ist, ist zunächst keine Interaktion mit anderen Views möglich. Um dieses Verhalten zu ändern, kann über die Property `passthroughView` des Popover Controllers eine Liste von Views gesetzt werden, die trotz sichtbaren Popovers auf Nutzereingaben reagieren.

26.3.3 Presentation Style für modale View Controller

Modale Views wurden im Buch einige Male benutzt. Für das iPad existiert nun die Möglichkeit, über die Property `modalPresentationStyle` des anzuzeigenden View Controllers zu definieren, ob der modale View den kompletten darunter befindlichen View verdeckt oder nur einen Teil davon. Mögliche Werte sind:

- `UIModalPresentationFullScreen`
- `UIModalPresentationPageSheet`
- `UIModalPresentationFormSheet`

`UIModalPresentationFullScreen` entspricht der vom iPhone bekannten Variante, bei der der modale View den darunter befindlichen View komplett verdeckt. Bei Verwendung von `UIModalPresentationPageSheet` erstreckt sich der modale View vom oberen

bis zum unteren Rand des Displays. Der View darunter ist in der Landscape-Orientierung aber im linken und rechten Randbereich noch zu sehen. In der Portrait-Orientierung liefern `UIModalPresentationFullScreen` und `UIModalPresentationPageSheet` das identische Ergebnis. `UIModalPresentationFormSheet` führt zur Anzeige des modalen Views in der Mitte. Wie die folgende Abbildung zeigt, ist der vorherige View auf allen vier Seiten noch zu sehen:

Bild 26.10: Der modale View-Stil `UIModalPresentationFormSheet`

26.3.4 Positionierung von Tool Bars

Bisher konnte die Tool Bar, wie sie auch im Chronos-Projekt verwendet wird, nur am unteren Bildschirmrand angezeigt werden. Außerdem war nur Platz für maximal fünf Tool Bar Items vorhanden. Auf dem iPad kann die Tool Bar wahlweise am oberen oder unteren Bildschirmrand platziert werden. Zudem steht durch die höhere Auflösung mehr Platz für Items zur Verfügung.

26.3.5 Input- & Accessory Views für Textkomponenten

In Kapitel 14 wurde besprochen, wie das Keyboard eingeblendet wird, wenn ein Textfeld oder *Text View* zum First Responder wird. Neu ist die Möglichkeit, alternativ zu den verschiedenen Tastaturen eigene *Input Views* zu erstellen. Dazu ist lediglich die Property `inputView` der Textkomponente auf den gewünschten Eingabe-View zu setzen. Dieser wird dann statt des Keyboards angezeigt.

Über diese Möglichkeit hinaus können Input Views für das iPad mit einem *Input Accessory View* versehen werden. Dabei handelt es sich um einen View, der am oberen Bereich des Input Views zusammen mit diesem angezeigt wird. Damit kann das Keyboard zum Beispiel um eine eigene Tool Bar erweitert werden. Zur Anzeige ist die Property `inputAccessoryView` der Textkomponente zu setzen.

Die folgende Abbildung zeigt einen eigenen Input-View zur Eingabe chemischer Formeln. Am unteren Rand des Views erlaubt ein Input Accessory View die Wahl des Periodensystems:

Bild 26.11: Custom Input View

Zum Erzeugen eines eigenen Input Views werden am besten zunächst Instanzvariable und Outlet vom Typ `UIView` in dem View Controller angelegt, der auch für den View mit der Textkomponente zuständig ist:

```
UIView *customInputView;
...
@property (nonatomic, assign) IBOutlet UIView *customInputView;
```

Anschließend wird wie gewohnt ein leeres Nib-File für den Input-View in Xcode erzeugt (*File > New File > User Interface > Empty XIB*). In das Nib-File wird ein View und auf diesen werden wiederum die benötigten Komponenten wie Buttons aus der Interface Builder-Library gezogen. Die Class Identity-Property des *File's Owners* im neuen Nib-File wird auf den View Controller mit dem neuen Outlet gesetzt. Anschließend können Outlet und View in der gewohnten Weise über eine Connection verbunden werden.

Zum Schluss wird das Nib-File im View Controller zum Beispiel in der Methode `viewDidLoad` geladen und die `inputView`-Property der Textkomponente auf den neuen View gesetzt:

```
[[NSBundle mainBundle] loadNibNamed:@"CustomInputView" owner:self
   options:nil];
textField.inputView = customInputView;
```

Für den `inputAccessorView` wird analog verfahren.

Damit beim Betätigen der Buttons auch die Kürzel der chemischen Elemente im Textfeld erscheinen, werden sie mit einer Action verbunden, die den Text auf dem Button in das Textfeld schreibt:

```
- (IBAction)addTextFromButton:(id)sender {
   UIButton *pressedButton = (UIButton *)sender;
   NSString *buttonText = pressedButton.titleLabel.text;
   NSString *textFieldText = textField.text;
   textField.text = [NSString
     stringWithFormat:@"%@%@",textFieldText, buttonText];
}
```

26.3.6 Weitere neue Klassen

Das SDK 3.2 enthält außer den besprochenen noch weitere neue Klassen. Einige besonders interessante sollen hier kurz erwähnt werden.

UIGestureRecognizer

Die abstrakte Klasse bietet zusammen mit ihren Subklassen `UITapGestureRecognizer`, `UIPinchGestureRecognizer`, `UIPanGestureRecognize`, `UISwipeGestureRecognizer`, `UIRotationGestureRecognizer` und `UILongPressGestureRecognizer` Unterstüzung bei der Erkennung von Gesten.

UITextChecker

Diese Klasse erlaubt die Überprüfung der Rechtschreibung und das Anbieten von Vervollständigungsoptionen für einen eingegeben Text.

UIMenuController

`UIMenuController` ermöglicht zusammen mit `UIMenuItem` das Zufügen von eigenen Menüeinträgen zum Menü, das beim Bearbeiten von Text im Standardfall die Optionen *Auswählen*, *Alles*, *Kopieren*, *Ausschneiden*, *Einfügen* und *Löschen* bietet.

Für weitere Informationen zu diesen und anderen Klassen sei auf den *iPad Programming Guide* verwiesen. Die Richtlinien für die Gestaltung des iPad-User Interfaces finden sich in den *iPad Human Interface Guidelines*. Beide Dokumente sind auf den Webseiten des iPhone Developer Centers erhältlich.

27 Geschäftsmodell iPhone

Gegen Ende des Buchs sollen in diesem etwas großspurig überschriebenen Kapitel einige Themen behandelt werden, die nichts mit Codierung und harten technischen Daten zu tun haben. Hier geht es zum Beispiel um die Frage, ob und wie sich mit der iPhone-Entwicklung Geld verdienen lässt und welche Schritte zur Vermarktung der App unternommen werden können. Eingeleitet wird das Kapitel mit einigen Zahlen und Fakten, die zwar schon bei Drucklegung des Buchs veraltet sein dürften, aber dennoch als Einstieg für die weitere Besprechung dienen sollen. Wenn Sie eigene Überlegungen anstellen möchten, sollten Sie versuchen, aktuelle Daten zugrunde zu legen.

27.1 Zahlen und Fakten

Mit ein wenig Internetrecherche lassen sich interessante Zahlen für das Geschäftsmodell iPhone herausfinden:

Apps im App Store:	150.000
Entwickler dieser Apps:	28.000
Verkaufte iPhones:	34 Millionen
Verkaufte iPod touch:	24 Millionen
iPhone-Nutzer in Deutschland:	1,5 Millionen

Die Zahlen sind natürlich je nach Quelle (auf deren Angabe hier bewusst verzichtet wird) mehr oder weniger zuverlässig, sie spiegeln aber sicherlich die Größenordnung im Februar 2010 wider.

27.2 Verdienstmöglichkeiten

Apps aus Spaß an der Programmierung zu entwickeln und weltweit unters Volk zu bringen macht viel Freude. Davon zeugen Tausende von Anwendungen ohne kommerziellen Hintergedanken, die im App Store verfügbar sind. Noch besser wäre es aber, wenn die Arbeit vernünftig bezahlt würde und so den Lebensunterhalt sicherte. Wie lässt sich mit der iPhone-Entwicklung Geld verdienen? Die nächstliegende Möglichkeit ist sicherlich, die Applikation im App Store nicht kostenlos anzubieten, sondern einen angemessenen Betrag dafür einzufordern. Darüber darf aber nicht vergessen werden, dass es noch andere Möglichkeiten gibt.

27.2.1 Werbung

Werbung auf mobilen Geräten ist ein Thema, das sicherlich in den kommenden Jahren noch weiter an Bedeutung gewinnen wird. Es gibt bereits eine ganze Reihe von Unternehmen, mit deren Software man Werbung in der eigenen App platzieren kann. Für kostenlose Anwendungen mit sehr großer Verbreitung kann das durchaus eine sehr einträgliche Form der Monetarisierung sein. Die folgende Abbildung zeigt eine App mit Werbung am unteren Rand:

Bild 27.1: App mit Werbung

Werbeanzeigen auf mobilen Geräten eröffnen außerdem völlig neue Perspektiven für die Werbebranche. Während sich beispielsweise Googles AdSense [URL-ADSENSE] zur Auswahl der Werbebotschaft (bzw. der Anzeige) bisher lediglich auf den Inhalt der Seite stützen konnte, stehen nun weitere Informationen wie etwa die aktuelle Position des Geräts zur Verfügung. Eine Anzeigeneinblendung für das nächstgelegene Restaurant zur Mittagszeit ist damit technisch kein Problem mehr.

Dass dabei Informationen aus der Privatsphäre des Benutzers übertragen werden, ist allerdings durchaus als problematisch zu betrachten. Die Datensammelwut einiger großer IT-Konzerne kennt bekanntlich keine Grenzen, und so wird sich neben anderen sensiblen Informationen wohl bald auch das gesamte Bewegungsprofil eines Benutzers auf den Servern wiederfinden. Wundern Sie sich also nicht, wenn auch die Litfaßsäule um die Ecke Sie demnächst ganz gezielt mit auf Ihre Interessen zugeschnittener Werbung anspricht …

Marktführer für mobile Werbung ist das Unternehmen Admob [URL-ADMOB], das von – wie könnte es anders sein – Google übernommen werden soll. Grundsätzlich können Sie Admob (und auch die unten genannten Konkurrenten) auf zwei Arten nutzen.

Entweder Sie zahlen dafür, dass andere Apps und mobile Webseiten Anzeigen für Ihre App schalten. Diese Variante fällt also eher unter den Aspekt »Marketing«.

Oder aber Sie stellen in Ihrer App eine Werbefläche zur Verfügung, um selber Geld mit der Werbung zu verdienen. Dort werden vom eingebetteten AdMob-Code automatisch Anzeigen geschaltet. Klickt der Benutzer auf die Anzeige (wobei unglücklicherweise Ihre App verlassen wird), erhalten Sie dafür eine Vergütung. Natürlich stört die Anzeige in der Anwendung. Allerdings sind die Anwender, genau wie bei Anzeigen auf Webseiten, bereit, diese Störung bis zu einem gewissen Grad in Kauf zu nehmen, um im Gegenzug dafür die App kostenlos zu bekommen.

Auch der Platzhirsch in Sachen Anzeigenschaltung im Internet, Googles AdSense, ist auf dem iPhone nutzbar. Das entsprechende Programm nennt sich *AdSense for Mobile Applications* und ist unter [URL-ADMOBILE] erreichbar. Der folgende Screenshot zeigt eine AdSense-Anzeige in der bekannten App *Shazam*:

Bild 27.2: AdSense-Werbung in App

27.2.2 Partnerprogramme

Eine weitere Einnahmequelle sind die zahlreichen Partnerprogramme, die im Internet existieren. Hier wird an Verkäufen von Reisen, Versicherungen und sonstigen Produkten mitverdient. Ihre App ist also quasi ein Client für das Backend des Partnerprogrammanbieters und kommuniziert mit dessen Servern.

Auch das Rekrutieren von Besuchern für eine Partnerwebseite kann Geld einbringen. Während sich diese Spielart auf dem iPhone mit seinem kleinen Display nur schwer durchsetzen kann, birgt das iPad, auf dem sich »normale« Webseiten vernünftig betrachten und vor allen Dingen bearbeiten lassen, dazu großes Potenzial. Viele Affiliate-Anbieter haben bereits eine API für die Zusammenarbeit mit anderen Webseiten. Oft kann diese auch für den Zugriff vom iPhone/iPad aus verwendet werden.

Auf eine Auflistung von einzelnen Programmen wird hier verzichtet, mit den Suchbegriffen »Partnerprogramm« oder »Affiliate« werden Sie mit der Suchmaschine Ihrer Wahl fündig.

27.2.3 Für Dritte

Fast unnötig zu erwähnen: Es gibt natürlich immer die Möglichkeit, Software für Dritte zu erstellen. Als Freiberufler vor Ort beim Kunden, als angestellter Mitarbeiter einer Firma oder auch als eigenes Unternehmen, das Auftragsarbeiten ausführt. Ob sich der Arbeitsmarkt nach dem momentanen App-Store-Hype in den nächsten Jahren für iPhone-Programmierer positiv entwickelt, hängt nicht zuletzt davon ab, ob das Gerät den Sprung in die Unternehmen schafft. Die Entwicklung von Individualsoftware, etwa zur Visualisierung wichtiger Unternehmenskennzahlen, könnte ein bedeutsames neues Betätigungsfeld werden.

27.2.4 Verkauf im App Store

Die folgenden Betrachtungen konzentrieren sich auf den Verkauf der App im App Store. Falls die Motivation für die Entwicklung Ihrer App in erster Linie finanzieller Natur ist (was gewiss keine Schande ist), dann sollten Sie einige Gedanken vor Beginn der Entwicklung anstellen. Letztlich geht es darum, abzuschätzen, ob sich die Arbeit und Mühe auch in barer Münze auszahlen werden. Wahrscheinlich haben auch Sie die unglaublichen Geschichten gehört oder gelesen, in denen Studienabbrecher innerhalb kürzester Zeit zu App-Store-Millionären wurden und noch immer werden. Es herrscht eine Goldgräberstimmung wie 1896 am Klondike. Wenn es bei Ihnen klappen sollte und mein Buch Ihnen dabei geholfen hat, gebe ich gerne die Kontonummer weiter. Um ehrlich zu sein, stehen die Chancen dafür allerdings nicht mehr ganz so gut. Nach dem Start des App Stores mit den ersten 500 Anwendungen, die der ausgehungerten iPhone-Gemeinde zum Download angeboten wurden, hat sich Erstaunliches getan.

Inzwischen stehen mehr als 150.000 (!) Apps zur Verfügung, und damit verteilt sich die Aufmerksamkeit des Publikums naturgemäß zunächst einmal deutlich breiter. Aber es kommt noch schlimmer. Weil niemand mehr diese Menge an Anwendungen überbli-

cken kann, gewinnen die Top-100-Listen für die verschiedenen Kategorien in iTunes ungeheuer an Bedeutung. Wer in der Liste ist, verdient gutes Geld. Für die anderen sieht die Lage nicht so rosig aus.

Der Verkauf von Programmen im App Store ist also zu einem Geschäft geworden, in dem, ganz genau wie in der Musikbranche, die Charts eine wichtige Rolle spielen. Um wirklich viel Geld zu verdienen, benötigen Sie einen Hit. Es mag sich nun jeder selbst eine Meinung darüber bilden, wie die Titel großer Musiklabels ihren Weg in die Top 100 schaffen. Im Abschnitt über Marketing werden Sie einige Ideen und Betrachtungen zu dem Thema finden. An dieser Stelle gehen wir vorerst davon aus, dass sich Ihre App trotz einer tollen Idee und einer technisch sauberen Umsetzung mit einer kleineren Bühne zufriedengeben muss.

27.3 Eine Milchmädchenrechnung

Nehmen wir an, Sie haben eine gute Idee für eine App und fragen sich, ob sich der Aufwand für die Entwicklung rechnen wird. Dazu stellen wir nun eine kleine Rechnung an.

Wenn man die Steuern vernachlässigt, ergibt sich der Erlös, den Sie mit der App erzielen können, aus dem Verkaufspreis, multipliziert mit der Anzahl der verkauften Einheiten, multipliziert mit 0,7 (weil Apple 30 Prozent einbehält):

Erlös = Preis x verkaufte Apps x 0,7

Wie Sie den Zahlen und Fakten entnehmen können, wurden bereits 58 Millionen Geräte mit iPhone OS verkauft. Wenn man nun davon ausgeht, dass jeder tausendste Benutzer die App kauft, kommt man auf 58.000 verkaufte Apps. Multipliziert mit einem zugrunde gelegten Preis von 2,99 Euro, abzüglich des 30-prozentigen Anteils, den Apple einbehält, ergibt sich so der mögliche Gewinn von 120.000 Euro. Eine Milchmädchenrechnung? Definitiv. Das Problem dabei ist offensichtlich die Annahme, dass jeder tausendste Benutzer die App kaufen wird. Der Wert ist vollkommen aus der Luft gegriffen. Es stellt sich die Frage, wie man zu einer besseren Einschätzung der potenziell verkaufbaren Einheiten kommen kann.

Dazu gilt es zunächst, die Größe der Zielgruppe herauszufinden. Wie viele Menschen dürfte die App interessieren? Wie schwer man an diese Informationen herankommt, ist abhängig von der Art der App. Zahlen von organisierten Personengruppen weltweit, wie die Zahl der Eishockeyfans in Berlin, die Zahl der Ärzte in Deutschland oder die Zahl der Chemiker, sind relativ leicht zu bekommen. Vereine, Verbände und Gesellschaften helfen sicherlich gerne weiter. Schwieriger wird es bei nichtorganisierten Gruppen wie etwa den bahnfahrenden Sudoku-Spielern. Für große kommerzielle Projekte kann man die benötigten Informationen notfalls auch käuflich von Marktforschungsinstituten erwerben. Auf jeden Fall handelt es sich bei der Zielgruppe um eine Größe, die Sie kennen sollten. Wir nehmen für die weiteren Ausführungen eine Zielgruppe von einer Million Menschen an.

Als Nächstes gilt es abzuschätzen, wie viel Prozent der Zielgruppe überhaupt ein Gerät besitzen, auf dem die Software läuft. Wir setzen einmal voraus, dass ein iPhone erforderlich ist, die App also nicht auf dem iPod touch läuft. Wie den genannten Zahlen und Fakten zu entnehmen ist, haben inzwischen etwa 2 Prozent aller Deutschen (1,5 Millionen) ein iPhone. Abhängig von der technischen Affinität der Zielgruppe kann diese Zahl aber noch verfeinert werden. Bei den Freiberuflern etwa, von denen viele in der IT oder in der Werbung arbeiten, wird der Wert vermutlich höher sein. Andere Personengruppen, wie etwa Rentner oder Schüler, sind sicherlich unterrepräsentiert. Eine Million Interessenten, von denen 2 Prozent auch wirklich die passende Hardware besitzen, ergeben lediglich 20.000 potenzielle Käufer. Das ist nicht viel.

Wie viele von denen werden die App wohl kaufen? Hier ist wieder Kaffeesatzlesen angesagt. Bei einer qualitativ guten Software und vor allen Dingen gutem Marketing könnten es vielleicht 3 Prozent sein. Selbst sehr erfolgreiche (kostenpflichtige) Apps erreichen kaum eine höhere Durchdringung.

Daraus ergeben sich dann die geschätzten verkauften Exemplare:

20.000 x 0,02 = 400

Werden diese mit dem Verkaufspreis multipliziert und der 30-prozentige Apple-Anteil abgezogen, erhält man um die 800 Euro:

400 x 2,99 x 0,7 = 837 Euro

Immer noch eine Milchmädchenrechnung? Leider ja. Einige der eingesetzten Werte sind einfach nicht vorherzusagen. Wenn Sie die Gesamtgleichung nun aber in eine Exceltabelle packen und ein wenig mit den Parametern spielen, werden Sie vielleicht doch ein Gefühl für die Spanne entwickeln, in der sich der Ertrag bewegen könnte. Ein paar im Netz recherchierte Verkaufszahlen vergleichbarer Apps bestätigen womöglich die Schätzung.

Letztlich bleibt aber ein schaler Nachgeschmack. Er lässt sich auch nicht durch die Tatsache vertreiben, dass Businesspläne für Firmengründungen mit ähnlichen Rechnungen bestritten werden. Andererseits ist eine schlechte Schätzung besser als gar keine und hilft womöglich, die hochgesteckten Erwartungen in realistischere Bahnen zu lenken.

Wie das Beispiel gezeigt hat, sollte die Zielgruppe nicht allzu klein sein. Eine geringe Anzahl an verkaufbaren Einheiten kann nur durch den Verkaufspreis kompensiert werden. Dieser wiederum steht in Relation zu dem Nutzen, den die App bietet, und wird nicht in beliebiger Höhe akzeptiert werden.

Ein Faktor, der nicht in die Rechnung eingeflossen ist, ist die im Moment steigende Anzahl der iPhone-, iPod touch- und iPad-Nutzer. Die Schätzung führt vielleicht in zwei Jahren zu einem besseren Ergebnis. Auch ist zu bedenken, dass das einmal geschaffene fachliche Know-how auch auf anderen Plattformen monetarisiert werden könnte. Viele erfolgreiche Apps sind für das iPhone, für Android-Geräte und für Blackberrys zu bekommen. Der Aufwand für eine Portierung ist natürlich geringer als für eine komplette Neuentwicklung.

27.4 Marketing

Viele Entwickler kennen die hinterletzte API-Funktion und können ihre IDE auch mit verbundenen Augen bedienen. Beim Thema Verkaufen allerdings herrscht Schweigen im Walde. Das muss nicht mit den persönlichen Fähigkeiten zusammenhängen, auch die Kunst des Vertriebs ist erlernbar. Letztlich bleiben den meisten Menschen aber eben nur 24 Stunden pro Tag, in denen man sich nicht mit allen Facetten des Softwarebusiness beschäftigen kann. Ähnlich wie das Thema Grafikdesign, das eigentlich Spezialisten vorbehalten sein sollte, ist auch Marketing eine Sache, die am besten an kompetente Partner vergeben wird.

Der Haken dabei ist, Sie haben es schon geahnt, das Ganze kostet eine Menge Geld. Für die großen Spieleentwickler oder andere bereits erfolgreiche Softwareschmieden ist das kein Problem. Für den Rest, also ca. 99 Prozent der Entwickler, leider schon. Gerade durch den Vertriebskanal App Store hat ein einzelner Entwickler plötzlich die Möglichkeit, ein Produkt ohne großen Aufwand international Millionen von Menschen anzubieten. Das wäre vor ein paar Jahren undenkbar gewesen. Lieschen Müllers kleines Sudoku-Spiel steht in direkter Konkurrenz zu Spieleklassikern, die im Fernsehen beworben werden.

Wie nun damit umgehen? Eins ist klar: Die Werbung darf möglichst nichts kosten. Bei den geringen Erträgen pro verkaufter App sind selbst günstige Werbemedien wie Google AdWords [URL-ADWORDS] eigentlich nicht tragbar. Auch klar ist, dass es ohne Werbung nicht geht. Zu groß ist die Gefahr, dass selbst eine gute Software angesichts der mehr als 200 neuen Apps, die im Moment täglich erscheinen, einfach untergeht. Dabei ist durchaus nicht sicher, dass Ihre App mit Marketing Erfolg haben wird. Ziemlich sicher ist aber, dass sie ohne geeignete Maßnahmen keinen haben wird. Der Trick ist also, viel Werbung zu machen, die wenig kostet.

Wie beim Design werden im Folgenden ein paar gängige Vorgehensweisen erläutert, die die Grundlagen dafür schaffen sollen, nicht im großen App-Store-Teich unterzugehen.

27.4.1 Der App-Store-Auftritt

Der Auftritt im App Store ist sozusagen die Visitenkarte für Ihre App. Hier sollte wirklich alles in Ordnung sein. Die Bemühungen in diesem Abschnitt konzentrieren sich im Wesentlichen darauf, Besucher auf diese Seite zu bekommen. Jeder potenzielle Benutzer, der Ihre App kaufen will, muss es über diese Seite tun. Wenn dann ein paar langweilige Screenshots mit einer nichtssagenden Beschreibung den Gast begrüßen, wird dieser schnell das Weite suchen. Die Konkurrenz lauert (über die Funktion *Kunden kauften auch*) meist nur einen Klick weit entfernt.

Name

Kurze Namen für Apps sind inzwischen ähnlich rar wie Domainnamen. DER Name für Ihre App ist womöglich schon vergeben. Die erste Idee für den Namen der Zeiterfassung war *Chronos*. Wie sich zeigte, war der Name schon bei Beginn der Arbeiten zu diesem

Buch vergeben. Glücklicherweise hat man sehr viele Freiräume bei der Vergabe des Namens, sodass sich immer eine Lösung finden lässt.

Ein guter Name sollte prägnant sein, neugierig machen und möglichst noch beschreiben, worum es bei der App geht. Reine Fantasienamen, die den Benutzer nötigen, auch die Beschreibung zu lesen, um herauszufinden, worum es überhaupt geht, haben es sicherlich ein wenig schwerer. Für die Zeiterfassung wurde der Name *ChronoLog* gewählt. Das *Chrono* deutet an, dass es hier in irgendeiner Form um Zeit geht. Der zweite Teil des Namens steht für die Logbuch-Funktion, also das Erfassen der Daten. Zusammen ergibt sich zusätzlich ein kleines Wortspiel: Chronolog, also in zeitlicher Reihenfolge. Der Name scheint auch kurz genug, um prägnant zu sein, und wirkt dank des großen »L« interessant. Um die Funktionalität der App ganz klar schon im Namen zu kommunizieren, wird der Zusatz *Die Zeiterfassung* angehängt:

- *ChronoLog – Die Zeiterfassung*

Icon

Mindestens genauso wichtig wie der Name ist das Icon für die App. Name und Icon sind die beiden zentralen Elemente, die darüber entscheiden, ob ein Benutzer sich die weitere Beschreibung für Ihre App überhaupt ansieht. Wie Sie den folgenden Abbildungen entnehmen können, sind das die beiden Informationen, die in der Suchergebnisliste und beim Blättern durch die Kategorien angezeigt werden. Und das gilt sowohl in der Anwendung *App Store* als auch in der iTunes-Variante des App Stores:

Bild 27.3: Suchergebnisliste in der App Store App

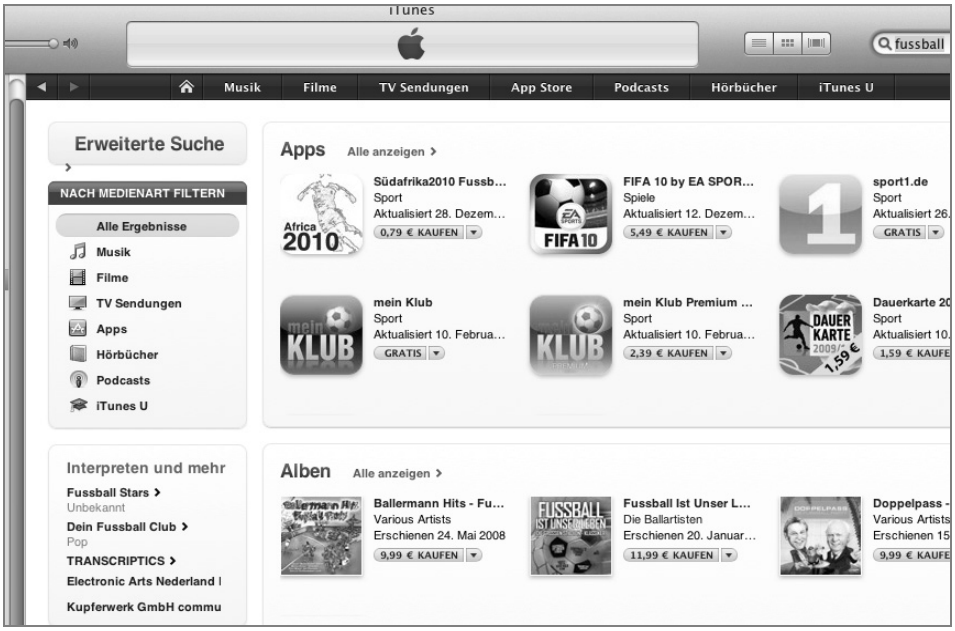


Bild 27.4: Suchergebnisliste in iTunes

Ganz sicher ist dies also eine der am schlechtesten geeigneten Stellen, um ein wenig Geld zu sparen. Falls Sie bereits über ein professionelles Logo, etwa für Ihre Webseite, verfügen, kann dies natürlich verwendet werden. Falls nicht, sollten Sie aber auf jeden Fall die etwa 200 Euro investieren und einen Grafiker mit der Aufgabe betrauen. Denken Sie immer daran, gegen wen Sie antreten – unter anderem die Großen der Spielebranche. Wahrscheinlich hat Ihre App kein ganzes Jahr Entwicklungszeit mit einem 50-köpfigen Team verschlungen. Aber man muss diese Tatsache ja nicht aller Welt mit einem billig aussehenden Logo kundtun. Im Gegenteil: Nutzen Sie die Chance, die die standardisierte Seite in iTunes bietet. Hier werden alle gleich behandelt.

Natürlich sollte das Icon den Zweck der App unterstreichen. Ergänzend zum Namen haben Sie nochmals die Möglichkeit, darauf hinzuweisen, um was es geht.

Beschreibung

Der Benutzer hat den Weg zu Ihrer Präsentation im App Store gefunden. Bei der Menge an Apps ist das keine Selbstverständlichkeit. Herzlichen Glückwunsch! Jetzt gilt es in der Beschreibung, Überzeugungsarbeit zu leisten. Warum sollte er ausgerechnet Ihre App kaufen?

Ein paar Vorschläge:

- Weil die Funktion Ihrer App in ein paar kurzen Sätzen gut erklärt wurde. Wie auf Webseiten mag der Benutzer hier keine Romane lesen.

- Weil ihn die aufgeführten positiven Zitate von anderen Benutzern oder, besser noch, Review-Bloggern überzeugt haben (»wenn die das gut fanden ...«).
- Weil er der Meinung ist, dass im Moment ein günstiger Zeitpunkt ist, die App zu kaufen, weil im oberen Bereich zu lesen ist, dass gerade eine zeitlich limitierte Sonderaktion läuft.
- Weil dem Benutzer neben der Auflistung der Features auch vermittelt wurde, was diese für ihn konkret bedeuten (Zeitersparnis, Sicherheit usw.).

27.4.2 Webseite

Zu jeder App gehört eine vernünftige Webpräsenz. Egal ob potenzielle Käufer über den entsprechenden Link im App Store oder auf anderem Weg auf Ihre Seite gelangen, begeistern Sie die potenziellen Nutzer für Ihr Werk. Während die Möglichkeiten im App Store begrenzt sind, kann hier die ganze Klaviatur der Webtechnologie gespielt werden. Weitere Screenshots, Videos, Newsletter oder Kundenmeinungen sind denkbare Optionen.

Da die Chance, im App Store gefunden zu werden, mit steigender Anzahl der Apps abnimmt, wird die Bewerbung des Programms über die Webseite immer wichtiger. Deshalb ist es ratsam, den Inhalt in für Suchmaschinen geeigneter Form zu präsentieren. Auf das Thema SEO (Search Engine Optimization) kann hier nicht weiter eingegangen werden, entsprechende Literatur gibt es zuhauf (z. B. »Suchmaschinenoptimierung & Usability« von Steven Broschart).

Absolutes Muss für die Webseite ist der Link zu Ihrer App im App Store. Er kann durch einen Rechtsklick auf das Icon in iTunes zum Beispiel aus der Suchergebnisliste kopiert werden. Apple stellt ergänzend dazu im *iPhone Dev Center* das bekannte Logo (*Available on the App Store*) zum Download zur Verfügung.

27.4.3 AdWords

Für Marketing zu bezahlen ist eigentlich der klassische Weg. Bei einem Artikel, der im Extremfall nicht mal einen Euro kostet, sollte man allerdings genau prüfen, ob sich die Ausgaben rechnen.

Die wahrscheinlich bekannteste Werbemöglichkeit im Internet ist Googles AdWords/AdSense. Dabei werden Anzeigen zu einem Produkt (etwa Ihrer App) auf Webseiten angezeigt, die vom Inhalt her potenzielle Käufer ansprechen. Sollte Ihre App etwa aus der Reisebranche stammen, dann wird sich der Link auf die Webseite zu Ihrer App in den Google-Anzeigen einer Reiseseite finden. Bei einem Klick auf die Anzeige wird die Gebühr für Google fällig. Der Preis ergibt sich im Prinzip als Ergebnis einer Versteigerung aller interessierten Werbenden. Natürlich können Sie in den AdWords-Einstellungen bestimmen, bis zu welchem Preis Sie mitbieten möchten. Allerdings macht ein zu kleines Limit keinen Sinn, weil die Anzeige dann einfach zu selten oder gar nicht geschaltet wird.

Ein realistischer Preis für einen Klick könnte, abhängig von den ausgewählten Keywords, beispielsweise 0,10 Euro sein. Wenn sich nun 5 Prozent, also jeder Zwanzigste, der über AdWords angelockten Besucher der Webseite auch wirklich zum Kauf Ihres Programms entschließt, dann haben Sie mit einem Werbeaufwand von 20 x 0,1 Euro = 2 Euro eine App für 1 Euro verkauft (davon gehen sogar noch die 30 Prozent für Apple und die Steuer ab). Offensichtlich kein Geschäft, das auf Dauer tragen wird. Natürlich kann nun auch hier wieder an den Zahlen gedreht werden. Vielleicht kaufen 10 Prozent der Besucher Ihre womöglich deutlich teurere App, und Sie kommen so in den rentablen Bereich. Auch mag die Beschränkung der Anzeigenschaltung auf mobile Webseiten die Rechnung ein wenig begünstigen. Dennoch – für die allermeisten Apps ist die Bewerbung über AdWords im Moment nicht sinnvoll. AdWords ist trotz des günstigen Preises hierfür schlicht zu teuer (bzw. das Preisniveau im App Store zu niedrig).

Direkt auf die Seite im App Store kann mit AdWords übrigens aus rechtlichen Gründen nicht verlinkt werden. Folglich ist für die Nutzer ein weiterer Klick nötig und damit eine weitere Hürde zu überspringen.

Wie Sie sich sicherlich vorstellen können, rechnen sich andere Formen der bezahlten Werbung für die typische Ein-Euro-App von Lieschen Müller noch weniger. Für große Firmen mit Apps in den Top-100-Listen sieht die Welt natürlich anders aus. Der Fall von einem Top-Listenplatz in die große Menge der schlecht sichtbaren 150.000 ist in finanzieller Hinsicht ein sehr tiefer. Die Unternehmen geben viel Geld für Werbung aus, und offensichtlich ist es gut angelegt.

27.4.4 Blogs und Review-Seiten

Unter den kostenlosen Werbemöglichkeiten sind zu allererst die zahlreichen Blogs und Review-Seiten zum Thema iPhone Apps zu nennen. Diese stellen mehr oder weniger interessante Apps vor, führen eigene Best-of-Listen und versuchen so dem Nutzer das unüberschaubare Angebot vorzufiltern. Eine Besprechung Ihrer App in einem oder besser mehreren dieser Blogs sorgt für Sichtbarkeit und sollte sich positiv auf die Verkaufszahlen der Anwendung auswirken.

Um in den Genuss einer Besprechung zu kommen, sollten Sie eine nette Mail verfassen, in der Sie auf das neue Produkt hinweisen, kurz erklären, warum es sich lohnt, diese App vorzustellen, und einen Promo-Code beilegen. Ein Promo-Code ist eine Art Gutschein, um die App kostenlos ausprobieren zu können. Man erhält maximal 50 Promo-Codes pro Version in *iTunes Connect*.

Außerdem sollten Sie einen Rezensionsvorschlag, also eine von Ihnen geschriebene Besprechung der App, etwa im Word-Format, zufügen. Im Internet ist Content bekanntlich King, und ein fertiger Artikel wird von so manchem Webmaster oder Redakteur gerne veröffentlicht. Auch wenn Sie dieses Vorgehen (zu Recht) für nicht ganz seriös halten, ist es leider gängige Praxis und damit ein Wettbewerbsnachteil, wenn Sie darauf verzichten. Womöglich wird der Redakteur des Blogs Ihren Vorschlag auch nur als Grundlage nutzen und entsprechend den eigenen Eindrücken ändern. Auf alle

Fälle sollte Ihre Rezension auch wie eine solche klingen und nicht wie eine Werbung für das Produkt (auch wenn das eigentlich beabsichtigt ist).

Einige Blogs zum Thema sind:

- Appsundco [URL-APPS&CO]
- iPhone App reviews.net [URL-IPARN]
- 148Apps [URL-148Apps]

Die Liste lässt sich nahezu endlos fortsetzen, unter den Suchbegriffen *iPhone*, *App* und *Review* werden Sie mehr Seiten finden, als Sie anschreiben können.

27.4.5 Pressemitteilungen

Ganz ähnlich ist das Vorgehen, um eine Pressemitteilung zu veröffentlichen. Hier sind Sie auf jeden Fall der Verfasser der Mitteilung und reichen diese zum Beispiel bei einer der folgenden Seiten ein:

- PRMac [URL-PRMAC]
- openPR [URL-OPENPR]
- PRWeb [URL-PRWEB]

Auch hier handelt es sich um eine willkürliche Auswahl, und die Liste lässt sich beliebig erweitern.

Obwohl diese Seiten nicht auf Apps spezialisiert sind, ist die Auswirkung einer solchen Mitteilung nicht zu unterschätzen. Im Gegensatz zu der angestrebten Besprechung im Blog können Sie relativ sicher sein, dass Ihre Mitteilung auch veröffentlicht wird. Das hat einen guten Grund: Viele dieser Seiten verlangen Geld für ihre Dienste. Die Spannbreite reicht von wenigen Euro bis zu mehreren Hundert Euro pro Mitteilung.

Da Pressemitteilungen das Futter für viele Blogs sind, machen Sie Ihre App auf diesem Umweg auf jeden Fall einer großen Anzahl von Nutzern bekannt.

27.4.6 Partnernetzwerke

Eine weitere interessante Möglichkeit, um zu zusätzlichen Downloads bzw. Verkäufen zu kommen, sind Partnernetzwerke, die Ihre App auf (mobilen) Webseiten oder innerhalb von anderen Apps bewerben. Um das Prinzip zu verstehen, können Sie sich zum Beispiel die kostenlose App *FTD* der Financial Times Deutschland aus dem App Store laden. In der App gelangen Sie unter *Mehr* > *App Store* zu einem View mit folgendem Aussehen:

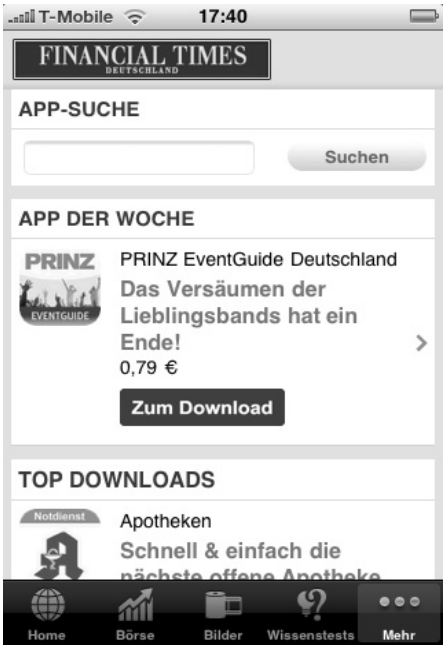


Bild 27.5: Partnernetzwerk in der FTD App

Wie auf dem Screenshot zu erkennen ist, werden hier andere Apps des Netzwerks beworben. Durch einen Klick auf den Button *Zum Download* gelangt man zur entsprechenden App im App Store.

Der Anbieter des Netzwerks erhält einen prozentualen Anteil vom Verkaufspreis der beworbenen App. Diese profitiert trotz der somit etwas reduzierten Erlöse womöglich von einem höheren Ranking in den Apple Top-Listen des Netzwerks. Als Beispiel für einen solchen Anbieter sei das Hamburger Unternehmen Apprupt GmbH [URL-APPRUPT] genannt. Die Lösung für das Partnernetzwerk nennt sich *apprupt Performance Network*.

27.4.7 Lite-Versionen, In App Purchase

Keiner kauft gerne die Katze im Sack. Unter den 150.000 ladbaren Anwendungen befindet sich eine Menge schwarzer Schafe. Während man vielleicht über die xte Version eines Views mit weißem Hintergrund, verkauft als Taschenlampe, noch streiten kann, sind andere Produkte eher als Betrug zu bezeichnen. Zum Beispiel werden freie Texte oder Bilder aus dem Web zusammengeramscht und für Geld verkauft. Zwar fallen nur wenige Benutzer auf die Apps rein, aber es lassen sich mit der gleichen Masche leicht einige Hundert Apps in den App Store bekommen. Die Masse macht's also. Apple wirft in schöner Regelmäßigkeit die betroffenen Apps aus dem Store.

Eine seriöse Möglichkeit, den Benutzer von der Qualität der eigenen App zu überzeugen, ist eine kostenlose Version mit eingeschränktem Funktionsumfang. Die Hemmschwelle für den Download der kostenlosen Grundversion ist sehr gering, und Sie (bzw.

Ihre App) haben die Möglichkeit, den Nutzer zu beeindrucken. Während vor dem iPhone SDK 3.0 dazu neben der normalen Version eine Lite-Version ausgeliefert werden musste, lassen sich nun kostenpflichtige Features durch das In-App-Purchase-Feature zukaufen. Inzwischen funktioniert das auch für zunächst kostenlose Apps.

Wenn Sie sich weiter mit dem wichtigen Thema App-Marketing beschäftigen möchten, sei auf das demnächst erscheinende Buch *iPhone and iPad Apps Marketing* [HUG10] hingewiesen.

27.5 Tracking und Statistiken

Die Tracking-Informationen, die in iTunes Connect erhältlich sind, sind bescheiden. Man erhält die Verkaufszahlen pro Tag/Woche/Monat und pro Region. Das ist zwar besser als nichts, aber so mancher Entwickler wünscht sich doch weitergehende Informationen, und zwar möglichst in Echtzeit. Ganz oben auf der Wunschliste stehen die folgenden Daten:

- Wie oft wird die App genutzt?
- Wie lange wird die App genutzt?
- Wo wird die App genutzt?
- Auf welchen Geräten wird die App genutzt?
- Welche iPhone-OS-Version ist auf den Geräten installiert?
- Welche Version der App wird benutzt?
- Welche Fehler traten auf?

Eine Reihe von Unternehmen bietet Werkzeuge, die diese Daten liefern. Ähnlich wie bei Google Analytics ein kleines Stück Javascript in die Webseite eingebaut wird, so wird bei den Tracking Tools ein Stück Code der betroffenen Lösung in die eigene App integriert. Dieser Code erfasst die gewünschten Daten und verschickt sie an den Server der Analytics-Lösung. Der Entwickler kann sich dort über einen Webaccount einloggen und die Daten einsehen oder herunterladen.

Ohne Zweifel können die oben genannten Informationen hilfreich bei der Identifizierung und Beseitigung von Fehlern sein. Aber Vorsicht: Was von Ihnen für eine notwendige Information zur Produktverbesserung gehalten wird, betrachten manch andere Zeitgenossen womöglich als neugierige Spyware. Auch wenn die erfassten Daten keinem Benutzer eindeutig zugeordnet werden können, wird hier doch ein Nutzerverhalten analysiert und ausgewertet. Sie sollten Ihre Nutzer deshalb über verwendete Tracking-Software informieren, wie das auch bei Webseiten üblich sein sollte.

Marktführer auf dem Gebiet der »Smartphone Application Analytics Provider« ist die Firma *Flurry*. Dabei handelt es sich inzwischen um einen Zusammenschluss von Flurry [URL-FLURRY] und einem ehemaligen Konkurrenten namens *Pinch Media*.

Das Tool von Flurry nennt sich *Flurry Analytics* und ist im Moment kostenlos auf der Flurry-Webseite erhältlich.

Nachwort

Gestatten Sie mir zum Schluss noch ein paar letzte Geleitsätze. Zunächst einmal hoffe ich, dass Ihnen das Buch gefallen hat. Noch wichtiger wäre mir aber, dass Sie nach dem Lesen des Buchs voller Ideen und Tatendrang sind, um ein eigenes Projekt zu realisieren.

Den Einsteigern in die Programmierung möchte ich für das erste Vorhaben den Ratschlag geben, sich nicht entmutigen lassen. Ihr Weg wird gepflastert sein mit Rückschlägen und Frust. Das ist ganz normal und liegt nicht an Ihnen. Jeder Fehler muss einmal begangen und verstanden werden, um ihn beim nächsten Mal zu vermeiden. Aber der Weg lohnt sich! Software zu entwickeln ist eine tolle Sache. Aus der eigenen Idee wird nach und nach ein echtes Programm, das nach den eigenen Vorstellungen funktioniert. Eine gelungene Mischung aus Kreativität und geistiger Herausforderung. Und das Beste ist, dass Sie damit Millionen Menschen in aller Welt erfreuen können. Bleiben Sie also dran und lesen Sie viel, viel Code. Das ist der beste Weg, um den Einstieg zu finden.

Auch den »gestandenen« Entwicklern möchte ich noch drei Hinweise mitgeben (die auf eigener leidvoller Erfahrung beruhen).

Planen Sie, bevor Sie beginnen. Während man bei der Erstellung von Web- oder Desktop-Anwendungen konzeptionelle Fehler im UI-Design und bei der Navigation meistens relativ leicht ausgleichen kann, ist das auf dem iPhone anders. Das geringe Platzangebot kann ein »Einfach-drauflos-Programmieren« wirklich in eine Sackgasse bringen. Es wäre schade, wenn Sie nach einiger Arbeit feststellen, dass Sie wichtige Funktionen, Ansichten oder Ähnliches vergessen haben und jetzt nur noch in unbefriedigender Art und Weise eingebaut bekommen.

Nutzen Sie eine Versionsverwaltung, auch wenn Sie allein arbeiten. Gerade am Anfang bringt man sich leicht in Situationen, in denen nichts mehr funktioniert, man aber nicht weiß, warum. Programmieren ohne Versionsverwaltung ist wie Akrobatik ohne Netz: sehr unangenehm, wenn etwas schief geht.

Begegnen Sie dem Thema Memory-Management mit der nötigen Aufmerksamkeit. Das Suchen der Memory-Leaks kann leicht genauso lange dauern wie die eigentliche Entwicklung. Versuchen Sie also erst gar keine einzubauen.

Falls Sie sich wissensdurstig noch weiter mit dem Thema beschäftigen möchten, habe ich noch ein paar Empfehlungen für Sie:

Das Buch *iPhone in Action* [AlA09] bietet einen guten Überblick über die Web-App-Entwicklung, die wir hier nicht behandelt haben. Zwar ist das Thema Web-Apps bei all der App-Store-Hysterie ein wenig in den Hintergrund getreten, für viele Anwendungsfälle bieten aber gerade sie die passende Lösung.

Viele Dinge versteht man besser, wenn man sie gesehen hat. Das Arbeiten mit den Connections im Interface Builder gehört sicherlich dazu. Auf der Seite *Pragmatic Bookshelf* [URL-PRAGPROG] finden Sie einige Video-Tutorials von Bill Dudney, die ihr Geld wirklich wert sind. Auch das Buch von Dudney und Adamson [DUAD09] kann sehr empfohlen werden.

Der Blog *Cocoa With Love* [URL-CWL] bietet ausgezeichnete Informationen zur Cocoa-Entwicklung mit Objective-C. Ebenso der Klassiker von Hilegas [HIL09] und das ganz neue Buch von Steinberg [STE10].

Noch ein guter Blog, allerdings mit anderer Ausrichtung ist *Mobile Orchard* [URL-ORCHARD]. Auf dieser Seite und dem ebenfalls dort zu findenden Podcast wird das gesamte iPhone-Business betrachtet.

So, das war's, was ich noch loswerden wollte. Wenn Sie Fehler finden, Kritik oder Anmerkungen äußern möchten, können Sie mich unter info@dirkkoller.de erreichen. Wichtige Informationen zum Buch werden auf meiner Homepage [URL-KOLLER] veröffentlicht.

Ich werde mir jetzt eine gute Zigarre genehmigen und warten, bis die ersten Konkurrenz-Zeiterfassungen im App Store auftauchen.

Legen Sie sich ins Zeug!

A Anhang

A.1 Präprozessor-Direktiven

Der Präprozessor bereitet den Sourcecode vor dem eigentlichen Kompilieren und Linken in einem maschinenfreundlichen Format auf. Dabei werden alle Präprozessor-Direktiven durch ausführbaren Code ersetzt.

Typische Aufgaben für den Präprozessor sind:

- Einfügen von Dateien
- Ersetzen von Makros
- Bedingte Übersetzung

Im Folgenden werden die einzelnen Direktiven vorgestellt. Alle Präprozessor-Direktiven beginnen mit einem Rauten-Zeichen (#).

A.1.1 #include

Diese Direktive wird benutzt, um den Inhalt einer anderen Datei in die eigene Datei einzubinden. Im Allgemeinen handelt es sich dabei um ein C- oder C++-Header-File, das sich selbst gegen Mehrfacheinbindung schützt.

A.1.2 #import

Mit `#import` wird wie mit `#include` ein Header-File importiert. Im Gegensatz zu `#include` wird das File allerdings nicht mehr als einmal eingebunden. Für Objective-C-Header ist die Verwendung von `#import` statt `#include` üblich.

A.1.3 #define

Mithilfe der `define`-Direktive können beispielsweise Konstanten definiert werden.

Beispiele:

```
#define MAXIMALANZAHL 25
#define STUNDENPROWOCHE 7*24;
#define PI 3.14
#define valid TRUE
```

Wo die Definition im Code erfolgt, ist egal. Üblich ist die Definition zu Beginn von Header oder Implementierung.

Die definierten Namen können im Code überall dort verwendet werden, wo ansonsten der zugewiesene Wert verwendet würde:

```
int ergebnis = MAXIMALANZAHL - 3;
```

A.1.4 #undef

Mit `undef` können zuvor durchgeführte Definitionen ungültig gemacht werden.

A.1.5 #ifdef/#else/#endif

Die Direktive fragt ab, ob eine bestimmte Definition existiert. Ist das nicht der Fall, wird die Anweisung zwischen `#ifdef` und `#endif` ignoriert bzw. ist für den Compiler nicht sichtbar.

Beispiel:

```
#ifdef DEBUG
   NSLog (@"Anzahl Kunden: ", kundenanzahl");
#endif
```

A.1.6 #if/#else/#endif

Mit `#if` können Anweisungen abhängig von dem auf `#if` folgenden Ausdruck in die Kompilierung übernommen werden.

Beispiel:

```
#if TARGET_IPHONE_SIMULATOR
  NSString *hello = @"Simulator";
#elif TARGET_OS_IPHONE
  NSString *hello = @"Device";
#else
  NSString *hello = @"Unknown";
#endif
```

A.1.7 #pragma mark

Die Direktive `#pragma mark` wird zur Organisation des Sourcecodes verwendet. Sie wirkt sich auf die Darstellung in der Methodenauswahlliste im Editor-Fenster von Xcode aus. `#pragma mark` gefolgt von einem Strich bewirkt die Anzeige eines Strichs in der Auswahlliste zur Abtrennung der Methodennamen.

Beispiel:

```
#pragma mark -
```

Der Trennstrich ist auf der folgenden Abbildung beispielsweise vor den Methoden im Abschnitt *Application Lifecycle* zu erkennen.

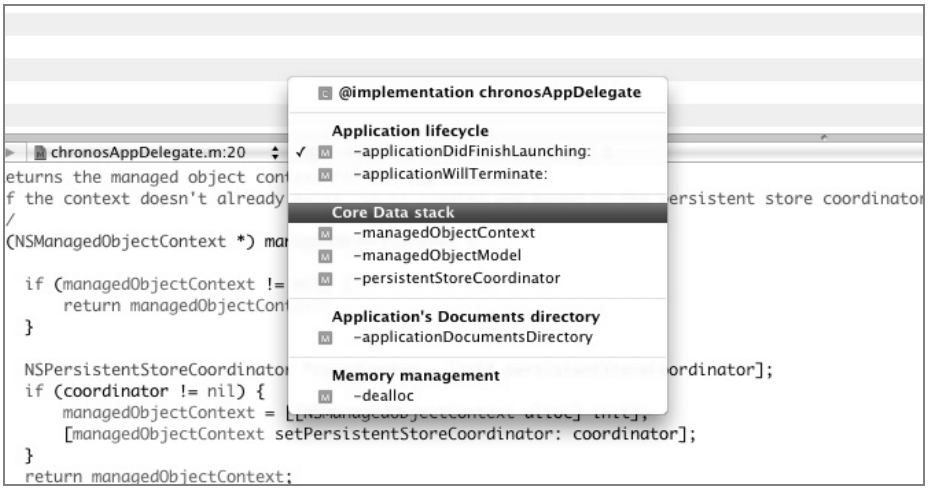


Bild A.1: Pragma mark-Direktive

Wird ein Text hinter `#pragma mark` angegeben, wird er fett in der Auswahlliste angezeigt.

Beispiel:

```
#pragma mark Core Data stack
```

Die Auswirkung der Direktive ist anhand der Markierung ebenfalls in obiger Abbildung zu erkennen.

A.1.8 #error

Mit `#error` kann ein Text als Fehlermeldung ausgegeben werden. Das ist natürlich nur mit einer Bedingung davor sinnvoll.

Beispiel:

```
#ifdef TEST
#error "Fehlermeldung"
#endif
```

Die folgende Abbildung zeigt die Fehlermeldung beim Build, wenn `TEST` definiert ist:

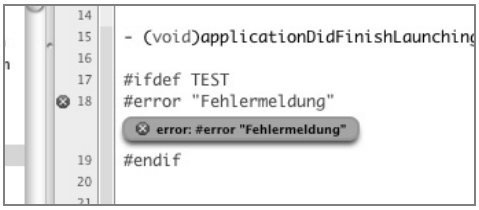


Bild A.2: Fehlermeldung mit #error

Glossar

Das Buch verwendet in der Softwareentwicklung allgemein übliches Vokabular. Lediglich einige spezielle Fachbegriffe, insbesondere aus der iPhone-Entwicklung, sollen hier gesondert erläutert werden.

App
App ist die Abkürzung für das englische Wort *Application* (zu deutsch Anwendung). Man versteht darunter kleine Programme für Smartphones. Für die iPhone-Plattform erhält man Apps ausschließlich über den App Store.

App Store
Eine Vertriebsplattform für Apps für die Geräte iPhone, iPod touch und iPad. Der App Store ist über das Programm iTunes auf Desktop-Rechnern oder die Anwendung App Store auf den mobilen Geräten erreichbar.

Callback
Callback-Methoden sind Methoden, die zwar vom Entwickler geschrieben, aber vom eigenen Code nie aufgerufen werden. Stattdessen werden sie aus einem bestehenden Framework aufgerufen.

Cocoa
Bei Cocoa handelt es sich um eine objektorientierte Entwicklungsplattform für Mac OS X. Sie besteht aus verschiedenen Objective-C-Frameworks, wie zum Beispiel dem Foundation-Framework.

File's Owner
Der File's Owner eines Nib-Files ist das Verbindungsglied zwischen Objekten, die grafisch im Nib-File mit Interface Builder angelegt wurden, und dem Anwendungscode, der diese Objekte benutzt. Im Normalfall handelt es sich dabei um einen View Controller.

Flick
Ein Flick bezeichnet eine Geste auf dem iPhone, bei der ein Finger auf dem Display in eine Richtung bewegt wird. Dadurch wird bei einigen UI-Elementen ein Scrollvorgang ausgelöst.

First Responder
Der First Responder ist das UI-Element, mit dem gerade interagiert wird, das also gerade den Fokus hat.

Hook
Ein Hook ist eine Schnittstelle im Programm, an der fremder Code eingebunden werden kann, also eine Art Erweiterungspunkt, den man für andere Programmierer hinterlässt.

iPhone SDK
Das Software Development Kit für die iPhone-Plattform. Mithilfe des Kits lassen sich Programme für iPhone, iPod touch und iPad erstellen und in einem Simulator testen. Im kostenlosen SDK sind zahlreiche im Buch beschriebene Werkzeuge und Frameworks enthalten.

Key Value Coding
Beim Key Value Coding werden die Properties eines Objekts nicht über `get-` und `set-`Methoden manipuliert, sondern mithilfe eines Strings, der als Key für die entsprechende Property fungiert. Dadurch wird das Setzen eines Wertes für eine Property von der Property selbst entkoppelt.

Nib-Datei
Als Nib-File wird ganz allgemein die mit dem Interface Builder erzeugte UI-Beschreibung bezeichnet. Dies betrifft sowohl binäre Dateien mit der Endung `.nib` als auch Dateien im XML-Format mit der Endung `.xib`.

Outlet
Eine Instanzvariable, im Allgemeinen in einem View Controller, die UI-Objekte in einem Nib-File referenziert. Durch das Outlet wird das Objekt aus der GUI-Beschreibungsdatei im Code verfügbar gemacht.

Pinch
Eine der Gesten, mit denen das iPhone bedient wird. Beim Pinch werden zwei Finger auf dem Display zueinander hin- oder voneinander wegbewegt. Dabei wird der Inhalt verkleinert oder vergrößert. Der Pinch sorgte für großes Aufsehen bei Vorstellung des iPhones.

Property-Listen
Property List Files sind Files mit der Endung `.plist`, in denen serialisierte Objekte gespeichert werden. Die Speicherung kann in XML oder einem binären Format erfolgen.

Tap
Ein Tap meint eine Berührung des iPhone-Displays mit dem Finger. Dabei wird das betreffende Element ausgewählt oder angeklickt. Der Tap entspricht einem Mausklick auf Desktop-Rechnern.

Window
Das Hauptfenster einer UIKit-Anwendung. Eine App kann aus mehreren Views, aber immer nur einem Window bestehen.

Xib-Datei
Das File mit der Endung `.xib` enthält die Beschreibung der Oberfläche im XML-Format. Es wird beim Kompilieren in das binäre File mit der Endung `.nib` umgewandelt. Obwohl die Datei auf `.xib` endet, spricht man (aus historischen Gründen) auch oft vom Nib-File.

Literaturverzeichnis

[HIL09] Aaron Hilegas: Cocoa Programming for Mac OS X. Addison-Wesley Longman, 2008

[STE10] Daniel H. Steinberg: Cocoa Programming. Pragmatic Programmers, 2010

[KOC09] Stephen Kochan: Programming in Objective-C 2.0. Addison-Wesley Longman, 2009

[Zar09] Marcus S. Zarra: Core Data. Pragmatic Programmers, 2009

[BEC03] Kent Beck: Extreme Programming – Das Manifest. Addison-Wesley, 2003

[GHJV01] Gamma, Helm, Johnson, Vlissides: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, 2001

[HUG10] Jeffrey Hughes: iPhone and iPad Apps Marketing: Secrets to Selling Your iPhone and iPad Apps. Que, 2010

[AlA09] Allen, Appelcline: iPhone in Action – Introduction to Web and SDK Development. Manning, 2009

[DUAD09] Bill Dudney, Chris Adamson: iPhone SDK Development. Pragmatic Programmers, 2009

[FOW00] Fowler: Refactoring – Improving the Design of existing code. Addison-Wesley, 2000

URL-Verzeichnis

[URL-148Apps] 148Apps
http://www.148apps.com

[URL-99DESIGNS] 99designs
http://99designs.com

[URL-ADMOB] Admob
http://www.admob.com

[URL-ADMOBILE] Google AdSense for Mobile Applications
http://www.google.com/ads/mobileapps

[URL-ADOBE] Kuler Adobe
http://kuler.adobe.com

[URL-ADSENSE] Google AdSense
www.google.com/adsense

[URL-ADWORDS] Google AdWords
http://adwords.google.de

[URL-APPRUPT] Apprupt GmbH
http://www.apprupt.com

[URL-APPS&CO] Apps & Co
http://www.appsundco.de

[URL-BUGZILLA] Bugzilla
http://www.bugzilla.org

[URL-CONNECT] iTunes Connect
https://itunesconnect.apple.com

[URL-CWL] Cocoa With Love
http://cocoawithlove.com

[URL-DESIGNENLASSEN] designenlassen.de
http://www.designenlassen.de

[URL-DEVCENTER] iPhone Dev Center
http://developer.apple.com/iphone

[URL-DEVCON] Apple Developer Connection
http://developer.apple.com

[URL-EULA] Apple End User License Agreement (EULA)
http://www.apple.com/legal/itunes/appstore/dev/stdeula

[URL-FLURRY] Flurry & Pinch Media
http://www.flurry.com

[URL-FOGBUGZ] FogBugz
http://www.fogcreek.com/FogBUGZ

[URL-GHUnit] GHUnit
http://github.com/gabriel/gh-unit

[URL-GNUSTEP] Gnustep
http://www.gnustep.org

[URL-GRAFFLE] OmniGraffle
http://www.omnigroup.com/applications/OmniGraffle

[URL-GTM] Google Toolbox for Mac (GTM)
http://code.google.com/p/google-toolbox-for-mac

[URL-INAPPSETTINGSKIT] InAppSettingsKit
http://www.inappsettingskit.com

[URL-INAPPSETTINGS] InAppSettings
http://bitbucket.org/keegan3d/inappsettings/wiki/Home

[URL-IPARN] iPhone Apps Review.net
http://www.iphoneappreviews.net

[URL-ISIMULATE] iSimulate
http://www.vimov.com/isimulate

[URL-ITEXT] iText
http://itextpdf.com

[URL-JSON] JSON
http://www.json.org

[URL-KOLLER] Homepage des Autors
http://www.dirkkoller.de

[URL-MANTIS] Mantis
http://www.mantisbt.org

[URL-METRICS] AdMob Mobile Metrics Report
http://metrics.admob.com

[URL-MONOTOUCH] MonoTouch
http://monotouch.net

[URL-OPENPR] openPR
http://www.openpr.de

[URL-ORCHARD] Mobile Orchard
http://www.mobileorchard.com

[URL-PRAGPROG] The Pragmatic Bookshelf
www.pragprog.com

[URL-PRMAC] prMac
http://prmac.com

[URL-PRWEB] PRWeb
http://www.prweb.com

[URL-SKYHOOK]
http://www.skyhookwireless.com

[URL-SQLITE] SQLite-Datenbank
http://www.sqlite.org

[URL-TOUCHJSON] TouchJSON
http://code.google.com/p/touchcode/wiki/TouchJSON

[URL-UIICONSET] iPhone UI Icon Set
http://www.eddit.com/shop/iphone_ui_icon_set

[URL-UNITY] Unity
http://unity3d.com

[URL-YOUTUBE] You Tube
http://www.youtube.com

Stichwortverzeichnis

Dr. Dirk Koller

iPhone-Apps entwickeln

Ein entscheidender Erfolgsfaktor des iPhones sind die Apps. Der erfahrene iPhone-Entwickler Dirk Koller demonstriert in diesem Buch, wie Sie Apps für das iPhone, den iPod touch und das neue iPad programmieren. Er beschreibt anhand einer Beispiel-App detailliert den Weg von der ersten Idee bis zu Upload und Vermarktung im App Store. Sie erfahren, wie Sie die App technisch sauber umsetzen, die Benutzeroberfläche entwerfen und mit Features wie z. B. Geodaten anreichern.

Von der Idee zur fertigen App

Dieses Buch zeigt den kompletten Weg einer iPhone-App: von der Grundidee über die nötigen Features bis zur Skizze der Benutzeroberfläche. Dann geht es an die Umsetzung: Autor Dirk Koller stellt die Entwicklungsumgebung Xcode, das Datenbank-Framework und die zentralen Klassen der Programmierschnittstelle des iPhone-Betriebssystems vor. Schritt für Schritt erläutert er die Entwicklung der iPhone-App und gibt Tipps dazu, wie Sie die App mit Features wie Geokoordinaten oder Kontaktdaten anreichern können. Entwicklern, die mit Java oder C++ vertraut sind, gibt er zudem einen Einblick in die iPhone-Programmiersprache Objective-C.

Einstellen in den App Store

Mit der Programmierung allein ist es noch nicht getan – schließlich entscheidet die Präsentation im App Store über den Erfolg der Applikation. Dirk Koller zeigt detailliert, wie Sie die App auf das Einstellen in den App Store vorbereiten und worauf Sie achten müssen, um den Qualitätskriterien von Apple zu genügen. Sie erfahren, wie Sie die App gründlich testen und überzeugend präsentieren. Darüber hinaus erhalten Sie Tipps für die erfolgreiche Vermarktung Ihres Programms.

Entwickeln für das iPad

Ausführlich geht der Autor auch auf die Programmierung für Apples jüngstes Kind, das iPad, ein. Das große Display des Tablet-Computers bietet ein ungeheures Potenzial für die Applikationsentwicklung. Dirk Koller zeigt, wie bestehende Apps an das iPad angepasst werden und wie Sie durch geteilte Bildschirmbereiche (Split Views) und Popovers (Fenster, die über andere Bereiche gelegt werden) die Darstellungsmöglichkeiten des iPads voll ausnutzen.

Aus dem Inhalt:

Über den Autor:

Dr. Dirk Koller ist seit zehn Jahren in der Softwareentwicklung als Entwickler, Berater und technischer Projektleiter tätig. Sein Tätigkeitsfeld ist die Java/Oracle-Entwicklung, zu der er regelmäßig Beiträge in Fachzeitschriften beisteuert. Seit dem Erscheinen des iPhones widmet er sich mit großer Begeisterung der Programmierung des Geräts. Er lebt und arbeitet in der Nähe von Frankfurt/Main.

Auf www.buch.cd:

Der komplette Quellcode der Beispiel-App „ChronoLog“

9 783645 600019

30,– EUR [D]
ISBN 978-3-645-60001-9