The Pragmatic Programmers

Cocoa-Programmierung

Der schnelle Einstieg für Entwickler

Deutsche Übersetzung von O'REILLY®

Daniel H. Steinberg

Übersetzung von Peter Klicman

Cocoa-Programmierung

Der schnelle Einstieg für Entwickler

Cocoa-Programmierung

Der schnelle Einstieg für Entwickler

Daniel H. Steinberg

Deutsche Übersetzung von
Peter Klicman

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:

O'Reilly Verlag
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

Die Originalausgabe erschien 2010 unter dem Titel
Cocoa Programming, A Quick-Start Guide for Developers bei Pragmatic Bookshelf, Inc.

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

Übersetzung und deutsche Bearbeitung: Peter Klicman
Lektorat: Volker Bombien, Köln
Korrektorat: Eike Nitz, Köln
DTP: Andreas Franke, SatzWERK, Siegen; www.satz-werk.com
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-613-6

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhaltsverzeichnis

Kapitel 1

Einführung

Während ich unser neues Haus gerade noch einmal abschließend besichtigte, sprach mich quer über die Straße eine Frau an: „Heute Abend findet unser jährliches Abendessen statt“, sagte sie, „kommen Sie doch vorbei und lernen Sie Ihre Nachbarn kennen!“

Ich folgte der Einladung und lernte alle meine neuen Nachbarn auf einmal kennen. Das Ganze ging schnell und war ein bisschen viel auf einmal. Es gab Unmengen an Informationen, von denen ich hinterher einige gebrauchen konnte. Aber hauptsächlich fühlte ich mich wohler, was meine neue Nachbarschaft anging, weil ich wusste, welche Fragen ich stellen musste, und die Leute kennengelernt hatte, die mir diese Fragen beantworten konnten.

Das ist auch das Ziel dieses Buches. Es ist kein Touristenführer, der alles aufführt, was man gesehen haben muss, wenn man Cocoa in nur einem Tag durchhechelt. Es ist auch kein umfassendes Lexikon, das jede API Klasse für Klasse und Methode für Methode aufführt. Es wurde konzipiert, um Sie durch die ersten Wochen und Monate mit Cocoa zu begleiten.

Es geht also um die Beantwortung derjenigen Fragen eines Programmierers, die solchen Fragen entsprechen wie wo man einen vernünftigen Kaffee bekommt, auf welchen Straßen man sich nachts sicher bewegen kann und auf welche Schule man seine Kinder schicken soll. Sobald Sie sich in Ihrer neuen Nachbarschaft eingelebt haben, werden Sie weitere Fragen haben, doch Sie werden wissen, wo und wie Sie nach Antworten suchen müssen.

1.1 Wir ziehen ein

Der Wechsel zu Cocoa ist wie der Umzug in eine neue Wohngegend. Sie müssen herausfinden, wo alles ist, und sich an die örtlichen Gepflogenheiten gewöhnen. Sie werden erkennen, dass einige Aspekte der Entwicklung von Cocoa-Anwendungen für Mac OS X stark dem ähneln, was Sie auch bisher schon gemacht haben, während sich andere Aspekte sehr fremdartig anfühlen.

In diesem Buch werden Sie ein Gefühl für Folgendes entwickeln:

- *Objective-C*: die Sprache der Cocoa-Entwicklung
- *Xcode*, *Interface Builder* und *Instruments*: die Werkzeuge der Cocoa-Entwicklung
- *Cocoa*: die Frameworks mit von Apple entwickelten vorgefertigten Klassen, die Ihren Anwendungen die Features und den Glanz Ihrer Lieblingsanwendungen unter Mac OS X verleihen

Was ist ein Framework, werden Sie fragen. Ein *Framework* ist eine Art Ordner, der eine zusammengehörende Gruppe von Ressourcen enthält. Sie können sich ein Framework als Bibliothek oder Paket vorstellen, aber es kann auch Bilddateien, Dokumentation, lokalisierte Strings und andere Konstrukte enthalten, die Sie später in diesem Buch noch kennenlernen werden. Sie werden in Kapitel 2, *Vorhandenes nutzen*, auf Seite 11 sehen, dass wir alle Ressourcen für die Programmierung von Webanwendungen einbringen, indem wir das `WebKit`-Framework in unser Projekt einbinden.

Frameworks sind Ihnen einerseits wohlbekannt, und anderseits doch ganz anders. Fürs Erste werden Sie gut damit fahren, sich Frameworks als Bibliotheken vorzustellen. Sie werden viele Ideen der Mac OS X-Entwicklung kennenlernen, die sehr nah an dem dran sind, was Sie bereits kennen. Sie werden versucht sein, die Sache so anzugehen, wie Sie es gewohnt sind. Tun Sie das nicht! Sie wollen nicht der Einzige sein, der auf der falschen Straßenseite fährt. Das wäre nicht gut für Sie, und es wäre auch nicht gut für die anderen Verkehrsteilnehmer.

Niemand mag den neuen Nachbarn, der davon erzählt, wie toll es dort ist, wo er herkommt. Das Gleiche gilt für OS X. Es ist nicht so, dass die alten Hasen kleinlich wären; sie gehen nur alles auf eine bestimmte Art und Weise an. Ihnen steht ein unglaubliches Potenzial zur Verfügung, native Mac OS X-Anwendungen schnell zu entwickeln, wenn Sie sich auf Objective-C einlassen, die Entwicklungswerkzeuge verwenden und die Vorteile des Cocoa-Frameworks nutzen. Sie werden sehr viel schnel-

ler sehr viel weiter kommen, wenn Sie den lokalen Gepflogenheiten folgen, statt gegen die Kultur anzukämpfen.

Verwenden Sie Objective-C. Sicher, Sie können Cocoa-Anwendungen auch in anderen Sprachen entwickeln, aber erst einmal sollten Sie die native Sprache erlernen. Neue Entwickler werden durch die verschiedenen Apple-Listen, die Dokumentation, Tutorials und den über Xcode zugänglichen Beispielcode sehr gut unterstützt. Ihre Fragen werden viel leichter beantwortet werden können, wenn Sie die Lingua franca der Cocoa-Entwicklung verwenden.

Was ist mit iPhone und iPad?

In diesem Buch geht es hauptsächlich um die Entwicklung für Mac OS X. Größtenteils handelt es sich um die gleichen Techniken, Tools und APIs, die Sie auch für iPhone und iPad verwenden. Es gibt einige Unterschiede, die ich in den iPhone-Kapiteln auch hervorhebe, aber dieses Buch konzentriert sich hauptsächlich auf Cocoa für den Desktop. Ich gehe davon aus, dass Sie sich leicht zurechtfinden werden, wenn Sie von der Cocoa-Entwicklung für Mac OS X auf die mobilen Plattformen wechseln.

Nutzen Sie die Werkzeuge, die Apple für die Cocoa-Entwicklung bereitstellt. In ihrer alten Umgebung haben Sie vielleicht ein Terminalfenster geöffnet und vi oder emacs zusammen mit gcc und gdb genutzt, oder vielleicht haben Sie eine IDE wie Eclipse verwendet. Für die Entwicklung von Cocoa-Apps verwenden Sie Xcode zum Schreiben, Kompilieren, Debuggen und Ausführen Ihres Codes. Sie verwenden Xcode auch zum Aufbauen, Generieren und Verwalten Ihrer Projekte. Sie nutzen Xcode sogar, um Ihre Datenmodelle zu erzeugen und zu bearbeiten. Sie werden den Interface Builder (IB) verwenden, um Ihre GUI zu entwickeln und die verschiedenen Komponenten zu verknüpfen. Sie werden Instruments nutzen, um die Performance Ihrer Anwendung zu verbessern. Sie finden die meisten der von Ihnen bevorzugten Kommandozeilenwerkzeuge in `/usr/bin/`, aber Sie sollten trotzdem Apples Entwicklungstools verwenden, wenn Sie eine Cocoa-App entwicklen.

Schließlich sollten Sie die fest eingebauten Frameworks einsetzen, so oft es geht. Bevor Sie darüber nachdenken, irgendwelchen Code zu schreiben, sollten Sie nachschauen, was Apple Ihnen zur Verfügung stellt. Ihr erster Impuls sollte immer sein, das zu nutzen, was es schon gibt.

Um diesen letzten Punkt besonders hervorzuheben, wird ihr erstes Projekt die Entwicklung eines einfachen Webbrowsers mit Apples `WebKit`-Framework[1] sein. Der Browser wird ein Textfeld besitzen, in das Sie eine URL eingeben können, sowie eine Webansicht, die die Webseite darstellt. Sie werden außerdem Vor- und Zurück-Buttons hinzufügen, um durch die bereits besuchten Seiten navigieren zu können. Da Sie sich dabei das `WebKit`-Framework zunutze machen, können Sie das alles erreichen, ohne irgendwelchen Code schreiben zu müssen.

Die Arbeit mit der neuen Sprache, den neuen Tools und den neuen APIs wird sich zuerst ein wenig fremd anfühlen. Sie sind mit ihnen nicht vertraut, weshalb die ersten Einfälle nicht immer die richtigen sind, doch im Nu werden Sie das eingeben, was Sie für den richtigen Namen der Methode halten, und herausfinden, dass es tatsächlich stimmt. Sobald Sie sich auf die Cocoa-Frameworks eingestimmt haben, werden Sie erkennen, dass Sie dem *Prinzip der geringsten Überraschung* folgen: Sie finden üblicherweise das vor, was Sie zu finden erwarten.

Joe fragt...

Richtet sich dieses Buch an mich?

Dieses Buch ist für erfahrene Programmierer gedacht, die mit der Mac- oder iPhone-Plattform nicht vertraut sind.

Sie verstehen grundlegende Programmierkonzepte, wissen aber nicht, wie sie in diesem Umfeld angewandt werden. Sie kennen eine Sprache, deren Struktur C ähnelt, doch Sie kennen Objective-C nicht. Sie verstehen die objektorientierte Programmierung, bloß in dieser Umgebung nicht. Sie wollen alle Techniken kennenlernen, die zur Arbeit mit den Cocoa-Frameworks nötig sind, doch es macht Ihnen nichts aus, die Dokumentation zu lesen, um die jeweiligen APIs zu erkunden, die Sie für Ihre Anwendung benötigen. Schließlich besitzen Sie einen Mac auf dem momentan Snow Leopard läuft.

Als Programmierneuling sollten Sie mit unserem Buch *Beginning Mac Programming* von Tim Isted beginnen, zu finden unter *http://pragprog.com/titles/tibmac/.*

1 Mir kam die Idee, mit diesem Beispiel zu beginnen, als ich Chris Adamsons „Zehn-Minuten-Browser"-Beispiel in *iPhone SDK Development* [DA09] redigierte.

Das hier ist auf keinen Fall ein allumfassendes Buch. Es behandelt nicht jede Ecke und jeden Winkel von Objective-C. Ich gehe nicht Klick für Klick alles durch, was Sie mit Xcode machen können, und wir behandeln auch nicht den ganzen Satz von APIs. Dieses Buch soll Sie auf den richtigen Weg bringen und gibt Ihnen dabei genug Informationen an die Hand, um Antworten auf neue Fragen zu finden.

1.2 Die Sprache erlernen

Beim Sprachenlernen in der Schule übersetzten Sie wahrscheinlich alles zwischen Ihrer Mutterspache und der neuen Sprache hin und her. Nach sehr viel Arbeit begannen Sie langsam, das Vokabular und die Grammatik zu meistern, und wurden mit den nativen Nutzungsmustern und Idiomen vertraut. Ohne es selbst zu bemerken, dachten Sie eines Tages in der neuen Sprache, während Sie sie sprachen oder lasen.

So läuft es auch bei Objective-C, der Sprache von Cocoa. Die Syntax ist anders, doch Vieles gleicht den Sprachen, die Sie bisher verwendet haben. Sie müssen darauf bedacht sein, sich von den Ähnlichkeiten nicht täuschen zu lassen, und sich gleichzeitig den Herausforderungen durch die Unterschiede stellen.[2] Sie werden sich schnell an die eckigen Klammern gewöhnen und an die Art, wie der Code strukturiert wird. Darüber hinaus müssen Sie sich mit den gängigen Mustern vertraut machen, die Cocoa-Programmierer üblicherweise verwenden.

In Kapitel 3, *Methoden und Parameter*, auf Seite 37, machen wir Sie mit dem Lesen von Objective-C vertraut. Wir beginnen mit Nachrichten (Messages), da das Senden von Nachrichten den Kern der Cocoa-Programmierung darstellt. Selbst erfahrene OO-Programmierer verlieren das schon mal aus den Augen. Wir neigen dazu zu glauben, dass es bei OO nur um Objekte geht.

Objective-C setzt auf C auf, verdankt aber viele seiner Ideen der Sprache *Smalltalk*. Es hilft, sich gelegentlich Alan Kays Mahnung aus dem Jahr 1998 an die Squeak-Mailingliste in Erinnerung zu rufen: Darin brachte er sein Bedauern darüber zum Ausdruck, dass er den Begriff *Objekte* geprägt hatte, weil er damit die Leute dazu verleitet habe, sich

2 Eines der großen Verkaufsargumente für Java war gleichzeitig sein Schwachpunkt: Die Syntax war vertraut, C-Programmierer konnten sehr leicht Java-Code schreiben. Doch sie schrieben oft Java-Code, der zu sehr wie C-Code aussah. Das gilt auch für viele andere Programmiersprachen, einschließlich Objective-C. Objective-C setzt auf C auf, d. h., Sie können reinen C-Code schreiben. Tun Sie das aber nicht!

auf die falsche Sache zu konzentrieren.[3] Kay erklärt, dass „der Schlüssel zur Entwicklung großer und erweiterbarer Systeme wesentlich vom Design der Kommunikation zwischen den Modulen abhängt, und nicht von ihren internen Eigenschaften und Verhaltensweisen".

1.3 Die Tools installieren

Stellen Sie sicher, dass die freien Entwicklungswerkzeuge installiert sind. Standardmäßig legt der Installer Entwicklungsanwendungen, Dokumentation, Beispiele und andere Dateien im `Developer`-Verzeichnis des Stammverzeichnisses ab. Zwar sind die Entwicklungswerkzeuge frei und auf den Installationsmedien von Mac OS X enthalten, werden standardmäßig aber nicht mitinstalliert. Überprüfen Sie das `Developer`-Verzeichnis und stellen Sie sicher, dass sie installiert sind. Starten Sie auch Xcode und wählen Sie `File > Get Info` oder drücken Sie ⌘ I, um das Xcode-Infofenster zu öffnen. Überprüfen Sie dort die Versionsnummer. Die Beispiele in diesem Buch verlangen mindestens Xcode 3.2.

Besorgen Sie sich die neuesten Entwicklertools (inklusive Beta-Releases), indem Sie der Apple Developer Connection (ADC) unter *http://developer.apple.com/* beitreten. Sie *sollten* der ADC beitreten. Es gibt eine kostenlose Mitgliedschaft, die Ihnen den Zugriff auf einen Großteil der Pre-Relase-Software ermöglicht. Es gibt auch kostenpflichtige Mitgliedschaften, die verschiedene zusätzliche Vorteile bieten.

Neu beginnen

Sie könnten Probleme haben, wenn Sie von einer früheren Mac OS X-Version auf Snow Leopard wechseln, oder wenn Sie von einer früheren Version der Developer Tools zur Developer Tools-Installation mit Xcode 3.2 wechseln. Diese Probleme manifestieren sich auf verschiedene Weise und treten häufig zutage, wenn man mit *.nib*-Dateien arbeitet.

Das lässt sich leicht beheben. Deinstallieren Sie zuerst die alten Developer Tools und führen Sie dann eine saubere Installation der neuen Developer-Tools durch – Problem gelöst.

3 *http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html*

Wir werden in diesem Buch einen Großteil unserer Zeit mit Xcode und dem Interface Builder verbringen. Sie werden Ihren Code in Xcode schreiben und Datenmodelle entwerfen, die wir später in diesem Buch verwenden werden. Sie werden den Interface Builder benutzen, um das Aussehen Ihrer Anwendung zu designen und Ihre visuellen Komponenten mit dem Code zu verknüpfen, der die Anwendungslogik enthält.

Obwohl wir diesen beiden Anwendungen den Großteil unserer Aufmerksamkeit widmen, erhalten Sie zusätzlich viele weitere Tools. Bevor Sie beispielsweise eine Anwendung für die Produktion freigeben, werden Sie sie eine Weile mit Profiling-Tools wie Instruments und Shark untersuchen wollen. Sie finden auch Audiotools, Grafiktools und eine Unmenge von Utilities. Wenn Sie die Developer-Tools installieren, wird auch eine Vielzahl von Kommandozeilenwerkzeugen installiert.

1.4 Die Frameworks erkunden

Wir werden ein wenig mit den Cocoa-Frameworks herumspielen. Wenn es geht, sollten Sie die von Apple bereitgestellten Objekte und Klassen verwenden, bevor Sie eigenen Code schreiben. Anfangs werden Sie viel Code schreiben, der etwas macht, was Sie in anderen Anwendungen unter Mac OS X gesehen haben. Ein erfahrener Cocoa-Entwickler wird sich Ihren Code ansehen, ein Gesicht ziehen und fragen: „Warum hast du nicht einfach …?“ So ungern Sie das vielleicht auch hören, seine zwei Codezeilen bewirken genau das Gleiche wie ihre 400 Zeilen. So ist das nun einmal.

Und dann, eines Tages, wird für Sie alles einen Sinn ergeben.

Das bedeutet nicht, dass Sie genau die beiden Codezeilen kennen werden, die Sie schreiben müssen. Doch nach 50 Zeilen wird Ihnen auffallen, dass Sie sich zu viel Arbeit machen. Sie werden wissen, dass es diese beiden Zeilen wahrscheinlich gibt. Wenn Sie selbst vielleicht nicht wissen, was zu tun ist, werden Sie zumindest wissen, wo Sie Hilfe finden.

Sie werden Ihren bevorzugten Weg in die Dokumentation finden. Ich neige dazu, die Options- und Befehlstasten (⌥ ⌘) zu drücken und die Klassen oder Methoden doppelt anzuklicken, um mir die Dokumentation für die Klasse bzw. Methode anzusehen. Sie können für einen Teil des Codes auch mit einem Rechtsklick oder einem Control-Klick ein kontextsensitives Hilfemenü öffnen, das Refaktorierungen, Links auf Definitionen und weitere verfügbare Optionen enthalten kann. Sie müssen sich diese Tastaturkürzel nicht merken. Sie erreichen die Dokumentation immer über die Xcode-Menüs.

Es gibt sehr gute Websites von Drittanbietern, und Sie können der *cocoa-dev*-Mailingliste beitreten, die Sie unter *http://lists.apple.com* finden. Sehen Sie sich zuerst ein wenig um, damit Sie ein Gefühl für die Liste bekommen. Und wenn Sie wirklich (vergebens) versucht haben, etwas selbst herauszufinden, dann trauen Sie sich und stellen Sie eine Anfrage, in der Sie beschreiben, was Sie getan haben und was Sie nicht verstehen oder erklärt bekommen möchten.

1.5 In diesem Buch

Dieses Buch hat vier Teile. Der erste Teil ist eine Einführung in Cocoa, Objective-C und die Tools. Der zweite Teil bringt Sie auf eine Ebene, auf der Sie mit den fundamentalen Techniken der Entwicklung einer Cocoa-Anwendung vertraut sind. Der dritte und der vierte Teil enthalten Material, das warten kann. Es ist wichtig, dass Sie die darin vorgestellten Konzepte verstehen, aber Sie müssen sich gedulden, bis Sie ein wenig Erfahrung gesammelt haben.

In diesem ersten Abschnitt werden Sie damit beginnen, Xcode und den Interface Builder zu verwenden. Sie werden sich an den Rhythmus des Wechsels zwischen beiden gewöhnen. Wir beginnen in Kapitel 2, *Vorhandenes nutzen*, auf Seite 11 mit einem Webbrowser, den Sie entwickeln werden, ohne eine einzige Zeile Code zu schreiben. Zwischen diesem Kapitel und Kapitel 9, *Anpassungen mit Delegates*, auf Seite 153 werden Sie die Grundlagen von Cocoa und Objective-C meistern.

In diesem ersten Teil hier werden Sie lernen, wie man Methoden erzeugt und aufruft, und die Angst verlieren, die Sie vielleicht vor der seltsam anmutenden Syntax mit ihren eckigen Klammern und Doppelpunkten haben. Sie werden Klassen in Xcode entwerfen und Objekte in Xcode und im Interface Builder erzeugen. Sie werden Eigenschaften und die Punktsyntax nutzen und – wann immer es geht – die automatische Garbage Collection verwenden. Sie werden die Regeln für das Reference-Counting für diejenigen Fälle kennenlernen, in denen sie den Speicher manuell verwalten müssen – z. B. bei iPhone-Apps. Sie werden in die beiden fundamentalen Muster der Cocoa-Entwicklung eingeführt: Delegates und MVC. Am Schluss holen wir einmal tief Luft und reimplementieren unseren Webbrowser für das iPhone.

Sie beginnen den zweiten Teil des Buches in Kapitel 11, *Notifikationen absetzen und abfangen*, auf Seite 191 mit einer neuen Anwendung, die darauf wartet, dass andere Anwendungen auf Ihrem Computer starten und beendet werden. Sie reagieren auf Benachrichtigungen, die der

Workspace sendet, in dem Ihre Anwendung läuft, und lernen, wie Sie eigene Benachrichtigungen senden. Sie werden Protokolle aufbauen, die beschreiben, welche Methode eine Klasse implementieren könnte, und Sie werden Delegates entwickeln, um Objekten ohne Subklassen zusätzliches Verhalten beizubringen. Sie werden Informationen in Schlüssel/Wert-Paaren in ein Dictionary schreiben – auf dieser Technik werden wir im dritten Abschnitt des Buches aufbauen.

Ein Schlüsselkonzept des zweiten Teils ist die Modularität: Wir zerlegen Methoden, Klassen und sogar Nibs in kleinere Teile. Am Ende dieses Abschnitts besitzen wir fünf *.nib*-Dateien für die Anwendung, um Hauptmenü, Fenster, Einstellungen und zwei Views zu organisieren. Einer davon ist ein benutzerdefinierter (*custom*) View. Sie lernen, wie man ihn zum Teil in Xcode und zum Teil im Interface Builder umsetzt. Ein Schlüssel zu dieser Modularität ist die Trennung von Modell, View und Controller. Das wird in unserem Kapitel zu benutzerdefinierten Views, bei unserer Arbeit mit Tabellen und in Kapitel 18, *Views wechseln*, auf Seite 289 hervorgehoben, wo wir den Benutzer im selben Fenster zwischen zwei Views hin- und herwechseln lassen.

Der dritte Teil konzentriert sich auf anspruchsvollere Möglichkeiten des Sendens und Empfangens von Nachrichten. Sie beginnen in Kapitel 19, *Key Value Coding*, auf Seite 303. *Key Value Coding* (KVC) erlaubt Ihnen zu entscheiden, welche Methode – basierend auf der Benutzereingabe – zur Laufzeit ausgeführt wird. KVC arbeitet perfekt mit Properties zusammen und erlaubt Ihnen, Objekte als Dictionaries zu betrachten und ihre Properties als Einträge in diesen Dictionaries. Sie werden sehen, was ich meine, wenn wir erst mal dort angekommen sind.

Sie können sogar noch einen Schritt weiter gehen und *Key Value Observing* (KVO) verwenden, um auf Änderungen eines Property-Wertes zu reagieren. Diese wenig formelle (*low-ceremony*) Notifikation ebnet die Bühne für Bindings und Core Data. Zu Beginn des Buches haben Sie erfahren, wie man den Interface Builder anstelle von Code benutzt, um Views aufzubauen. Dieser Abschnitt zeigt Ihnen, wie Sie einen Teil Ihres Controller-Codes und sogar einen Teil des Modellcodes entfernen können.

Das Buch beginnt mit der Entwicklung einer Anwendung fast ohne Code. Sie verbringen dann einen Großteil des Buches damit, verschiedene Coding-Techniken zu meistern. Gegen Ende von Kapitel 23, *Kategorien*, auf Seite 379 werden Sie wieder weniger Code schreiben. Diesmal lernen Sie aber anspruchsvollere Techniken kennen, die Ihnen ermög-

lichen, leistungsstarke und flexible Anwendungen zu entwickeln, indem Sie nur genau den Code schreiben, der auch wirklich benötigt wird.

Der vierte Teil ist kurz und führt in Blöcke und zwei Methoden für den Umgang mit Nebenläufigkeit (*concurrency*) ein. In Kapitel 24, *Blöcke*, auf Seite 395 lernen Sie mit Blöcken ein neues Konstrukt kennen, das in Snow Leopard eingeführt wurde. Alle Techniken, die Sie bis zu diesem Punkt kennengelernt haben, gelten unverändert auch für Code, den Sie bei Leopard einsetzen.[4] Wenn Sie aber in Ihrem Code Blöcke nutzen, ist er für Leopard oder das iPhone nicht geeignet. Sie sind auf Snow Leopard (und höher) beschränkt.

Ich habe mit mir gerungen, ob ich Nebenläufigkeit in diesem Buch behandeln soll oder nicht. Sie ist eine komplizierte Angelegenheit, wenn man sie richtig machen will, und man könnte ein ganzes Buch zu diesem Thema schreiben. Mein Rat ist einfach: Verwenden Sie Threads nicht direkt. Versuchen Sie es zuerst mit Operation-Queues. Wenn das Ihren Bedürfnissen nicht entspricht und Sie mit Snow Leopard (oder höher) arbeiten, gehen Sie einen Schritt weiter und arbeiten Sie mit Dispatch-Queues.

Wie Sie sehen, konzentriert sich das Buch auf Techniken, nicht auf APIs. Gegen Ende des Buches sollten Sie gut aufgestellt sein, um jede beliebige Cocoa-Aufgabe in Angriff nehmen zu können. Selbst wenn Sie nicht genug über das spezifische Problem erfahren haben, das Sie lösen müssen, sollten Sie genug Erfahrung haben, um die richtige Stelle in der Apple-Dokumentation aufzuspüren und herauszufinden, *was* Sie *wie* tun müssen, um zum Erfolg zu gelangen.

Willkommen in der neuen Nachbarschaft. Ich freue mich, dass Sie hier sind. Ich kann es gar nicht erwarten, die coolen Anwendungen zu sehen, die Sie entwickeln werden, nachdem Sie dieses Buch gelesen haben.

4 Die APIs können sich unterscheiden, aber die Techniken sind gleich.

Kapitel 2

Vorhandenes nutzen

Beginnen wollen wir mit der Entwicklung eines Webbrowsers, und zwar so:

Wir werden diesen Browser entwickeln, ohne eine einzige Zeile Code zu schreiben. Er wird nicht besonders raffiniert sein, aber er wird funktionieren. Sie werden URLs eingeben können, um ihre Lieblingsseiten zu laden. Sie werden zwischen den Seiten navigieren können, indem Sie den Links folgen (genau wie in einem normalen Browser), und Sie werden Buttons nutzen können, um sich in Ihrer Browser-History vor- und zurückzubewegen.

In diesem Kapitel werden wir den Interface Builder und standardmäßig verfügbare Komponenten verwenden, um unseren Browser zu bauen. Wir verwenden ein paar Buttons, ein Textfeld und eine spezielle Komponente zur Darstellung von Webseiten aus einer Library. Wir ziehen diese Komponenten in unser Fenster, ordnen sie unserem Wunsch entsprechend an und stellen sicher, dass sie sich visuell richtig verhalten, wenn man das Fenster vergrößert oder verkleinert. Wir benutzen dann den Interface Builder, um das von uns gewünschte Verhalten zu aktivieren. Sobald alles so gut funktioniert, wie wir es im Moment hinkriegen können, werfen wir einen Blick hinter die Kulissen.

2.1 Ein Projekt in Xcode anlegen

Während wir in diesen ersten Kapiteln ständig zwischen IB[1] und Xcode hin- und herwechseln, entwickeln Sie ein Gefühl dafür, wofür die beiden gedacht sind. Sie werden Xcode hauptsächlich als IDE nutzen, um an Ihrem Code, Ihrem Datenmodell und Ihrem Projekt zu arbeiten. Die Arbeit, die Sie mit Xcode erledigen, sollte Ihnen vertraut sein, wenn Sie mit anderen IDEs arbeiten.

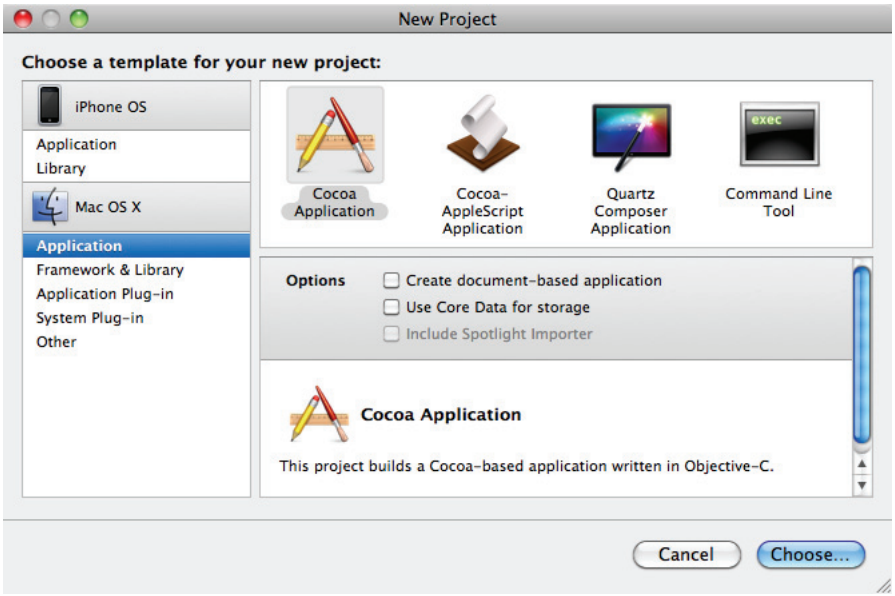
Wie Sie wohl schon erraten haben, entwickeln Sie Ihre GUI mit dem Interface Builder. Sie werden ihn benutzen, um Elemente der GUI mit Datenquellen und anderen GUI-Elementen zu verknüpfen, sowie mit im Code beschriebenen Methoden. Bei diesem ersten Beispiel werden Sie den Großteil der Zeit im Interface Builder verbringen.

Auch wenn Sie keinen Code für das SimpleBrowser-Beispiel schreiben, müssen Sie ein Projekt in Xcode anlegen. Starten Sie Xcode und legen Sie ein neues Projekt an, und zwar entweder über ⌘ ⇧ N oder über `File > New Project`.[2]

Ihnen wird eine Vielzahl von Optionen für Projektvorlagen präsentiert, mit denen Sie Anwendungen für das iPhone oder für Mac OS X entwickeln können. Wählen Sie `Mac OS X > Application > Cocoa Application`. Lassen Sie die Checkboxen für die Entwicklung einer dokumentenbasierten App und für die Verwendung von Core Data so, wie sie sind. Klicken Sie dann den *Choose*-Button an oder drücken Sie einfach ↵.

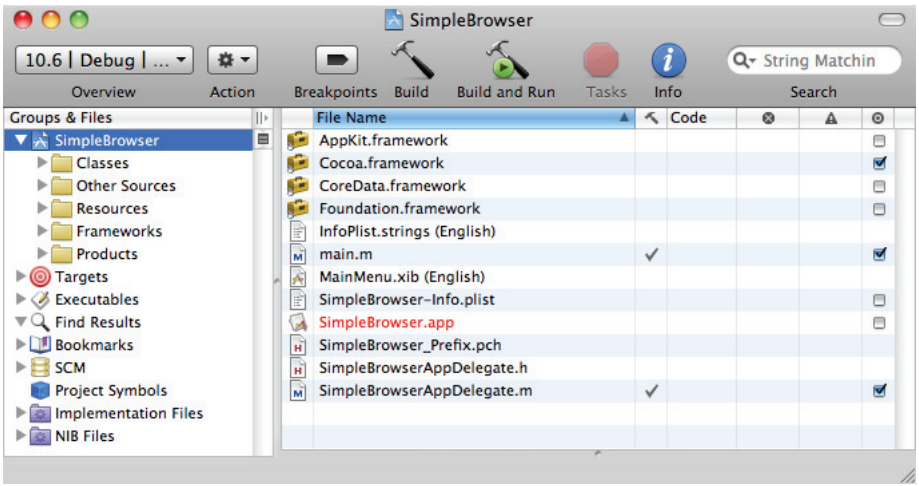
1 Das ist die Abkürzung für „Interface Builder“.

2 Sie finden Xcode im Verzeichnis `/Developer/Applications`. Sie werden es so oft brauchen, dass Sie es auf das Dock ziehen sollten. Denken Sie auch daran, dass Sie immer Spotlight benutzen können, indem Sie das Vergrößerungsglas auf der rechten Seite der Menüleiste anklicken.



Ich habe das Projekt unter `~/Dev/UsingWhatsThere/` als *SimpleBrowser* gespeichert. Sie können natürlich einen anderen Namen und einen anderen Ort wählen.

Ihr neues Projekt sollte etwa so aussehen:



Es wird automatisch eine ganze Menge Infrastruktur für Sie angelegt. Zunächst einmal ignorieren wir die meisten dieser Dateien. In diesem Kapitel erledigen Sie alles in der Datei `MainMenu.xib`.

Unsere SimpleBrowser-Anwendung kann im Moment noch nichts, aber wir können sie schon testen. Klicken Sie auf *Build & Run*, und nach einem Augenblick des Kompilierens und Verlinkens erscheinen ein leeres SimpleBrowser-Fenster und eine Menüleiste mit den Standardmenüs und Menüpunkten. Sie werden auch bemerken, dass `SimpleBrowser.app` in Xcode nicht mehr rot markiert ist, da die Datei nun existiert. Beenden Sie den SimpleBrowser[3], damit Sie mit dem Aufbau der Benutzerschnittstelle beginnen können.

2.2 Das Aussehen mit dem Interface Builder entwickeln

Wählen Sie die Datei `MainMenu.xib` (English) im Xcode-Fenster mit einem Doppelklick aus. Das ist ihre *.nib*-Hauptdatei. Eine *.nib*-Datei enthält die gesamte Arbeit, die Sie im Interface Builder vornehmen. Im Interface Builder erzeugen, konfigurieren und verknüpfen Sie Objekte mithilfe grafischer Werkzeuge. Jede *.nib*-Datei ist im Grunde ein schockgefrorener Objektgraph, der zur Laufzeit wiederhergestellt wird. Diese erste Anwendung verwendet nur eine *.nib*-Datei. Im weiteren Verlauf dieses Buches werden Sie aber kompliziertere Anwendungen mit mehr als einer *.nib*-Datei entwickeln.

.nib-Dateien werden eigentlich in einem Binärformat gespeichert und besitzen die Erweiterung `.nib`. Während der Entwicklung werden sie aber in XML-Dateien abgelegt, damit sie sich besser mit Versionskontrollsystemen vertragen. Sie sollten diese Dateien aber nur im Interface Builder bearbeiten und das Persistenzformat unangetastet lassen. Der einzige für Sie erkennbare Unterschied besteht darin, dass die XML-Versionen die Erweiterung `.xib` besitzen. Trotz dieser Änderung werden sie meist immer noch als *.nib*-Dateien bezeichnet (gesprochen „nib“, nicht „en-i-be“), auch wenn einige Leute `.xib`-Dateien als „zibs“ bezeichnen.[4]

Unser SimpleBrowser besteht aus einem einzelnen Fenster mit Standardkomponenten. Sie werden das `MainMenu-`.nib mit einem Web-View für das Rendern von Webseiten, einem Textfeld für die Eingabe von Adressen und zwei Buttons für die Navigation füllen.

---

3 Durch das Schließen des Fensters wird die Anwendung nicht beendet. Sie müssen den Task in Xcode beenden, indem Sie im aktiven SimpleBrowser ⌘Q drücken oder es über das Menü beenden.

4 Der Name *nib* leitet sich aus dem Akronym für den NeXT Interface Builder ab. Der Interface Builder und das Framework, aus dem Cocoa entstanden ist, wurden bei NeXT Computer entwickelt. Wenn Sie ein Release erzeugen, werden die `.xib`-Dateien zu `.nibs` kompiliert.

Joe fragt...

Sollten wir nicht Code schreiben?

Alles, was Sie mit dem Interface Builder machen können, lässt sich auch mit (Programm-)Code erreichen. Generell gilt aber, dass alles, was sich im Interface Builder erledigen lässt, auch im Interface Builder erledigt werden sollte, und nicht mit Code.

Wenn Sie sich strikt an diese Regel halten, werden Sie beim Programmieren nur anwendungsspezifischen Code entwickeln (und keinen allgemeinen). Üblicherweise ist es nichts Besonderes, Buttons in Ihrer Anwendung zu platzieren und zu instanziieren – also nehmen Sie dafür den Interface Builder.

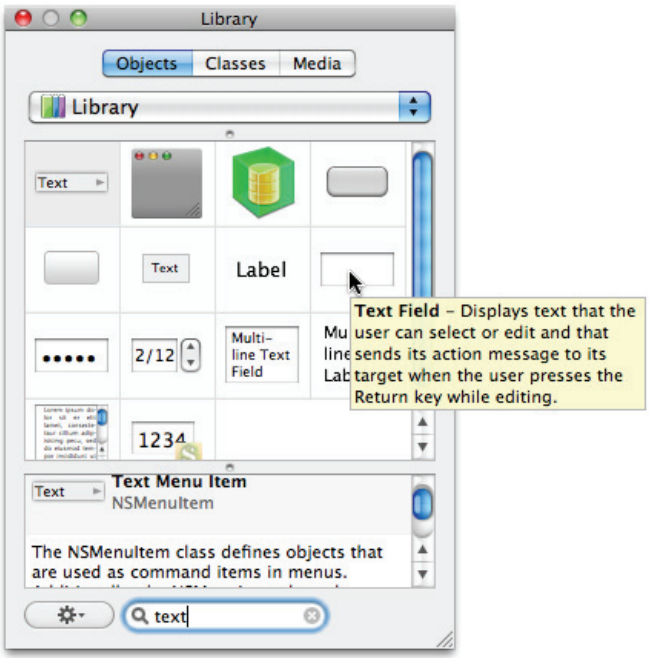
In diesem Kapitel verwenden wir nur gängige visuelle Komponenten wie Fenster, Buttons, Textfelder und Web-Views. Wir müssen keine eigenen Klassen entwickeln, da Apple sie bereits für uns entwickelt und (die meisten) in die Cocoa-Frameworks gepackt hat. Wir erzeugen Instanzen dieser visuellen Elemente und ordnen Sie im Interface Builder passend an.

Am Ende des Kapitels werden Sie sehen, dass wir den Interface Builder auch benutzen können, um einen Button mit der entsprechenden Aktion zu verknüpfen. Der von Apple bereitgestellte WebView weiß bereits, wie man zu einer angegebenen URL gelangt und sich in der Browser-History vor- und zurückbewegt. Wir nutzen im Moment dieses fest eingebaute Verhalten und schieben die Entwicklung von Code auf, bis wir das Verhalten des Browsers anpassen möchten.

Sie haben den Interface Builder gestartet, indem Sie `MainMenu.xib` doppelt angeklickt haben. Sie sollten nun mehrere Fenster sehen. Eines enthält die Informationen zu Ihrer Menüleiste; ein anderes heißt *MainMenu.xib* und enthält Dinge wie *File's Owner* und *First Responder*. Sie interessiert das leere Fenster mit dem Titel *SimpleBrowser*. Das ist das Fenster, das all unsere visuellen Elemente enthalten soll.

Öffnen Sie die Library über ⌘⇧L oder `Tools > Library`. In der Library finden Sie alle Objekte, die Apple zur Verfügung stellt, sowie die Objekte, die Sie in Xcode entwickeln. Sie können sich zu den gewünschten Elementen bewegen oder das Suchfeld am unteren Rand des Fensters benutzen, um das Ergebnis zu filtern. Um sich die von Apple bereitgestellten Komponenten anzusehen, aktivieren Sie den

Objects-Reiter am oberen Rand und wählen in der Drop-down-Liste gleich unter dem Reiter den Punkt *Library*. Ich habe hier *text* eingegeben, um das Textfeld zu finden, das ich zur Eingabe der URL verwenden möchte.

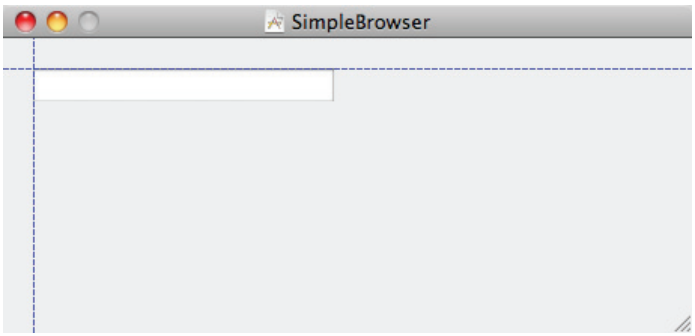


Sie können das mittlere Panel des Library-Fensters mit der rechten Maustaste anklicken, um es zu konfigurieren. Ich habe mich für die Darstellung der Icons entschieden, aber Sie können sich auch die Icons und Labels oder die Icons und die Beschreibungen der GUI-Elemente anzeigen lassen. Egal, welche Ansicht Sie wählen – Sie können sich (wie in der Abbildung zu sehen ist) immer zusätzliche Details anzeigen lassen, indem Sie die Maus über eine der Komponenten bewegen.

Klicken Sie das Textfeld in der Library an und ziehen Sie es in das leere Fenster. Während Sie das Textfeld im Fenster bewegen, sehen Sie blaue Leitlinien, die Ihnen dabei helfen, die Elemente entsprechend den Human Interface Guidelines von Apple zu positionieren.

Platzieren Sie das Textfeld mithilfe der linken und oberen Leitlinien in der oberen linken Ecke. Möglicherweise müssen Sie die Größe des Fensters anpassen, indem Sie die untere rechte Ecke des Fensters nach rechts unten ziehen. Sie benötigen genügend Raum, um rechts

neben dem Textfeld zwei Buttons zu platzieren. Sehen Sie sich den Browser auf der ersten Seite dieses Kapitels noch einmal an, damit Sie wissen, was wir hier gerade aufbauen.



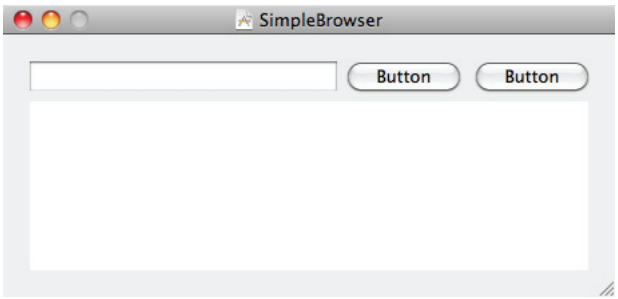
Wechseln Sie zur Library und löschen Sie das Wort *text* aus dem Suchfeld. Aktivieren Sie den *Objects*-Reiter und bewegen Sie sich in der Liste darunter zu `Library > Cocoa > Views & Cells > Buttons`. Sie sehen ein Dutzend verschiedene Buttons. Alle sind Instanzen der Klasse `NSButton`, die aber in unterschiedlichen Situationen eingesetzt werden. Wenn Sie einen Button anklicken, erscheint am unteren Rand des Fensters ein Text, der Sie darüber informiert, was für eine Art Button Sie gerade gewählt haben. Sie brauchen den *Push*-Button.

Klicken Sie den *Push*-Button in der Library an und ziehen Sie ihn rechts neben das Textfeld, das Sie eben in Ihrem Fenster platziert haben. Sie sehen blaue horizontale Leitlinien, mit deren Hilfe Sie sicherstellen können, dass Sie sich auf einer Höhe mit dem Textfeld befinden. Wenn Sie den Button näher an das Textfeld bewegen, erscheint eine vertikale blaue Linie links neben dem Button, die Ihnen anzeigt, dass sich die Elemente im richtigen Abstand befinden.

Wechseln Sie noch einmal zur Library, wählen Sie einen weiteren *Push*-Button und ziehen Sie ihn rechts neben den anderen. Der obere Bereich des Fensters sollte nun ein Textfeld und zwei Buttons enthalten. Wir werden noch Anpassungen an diesen drei Elementen vornehmen, ihre Größe und Position können Sie also später noch korrigieren.

Sie müssen noch ein weiteres Element platzieren. Wechseln Sie in die Library und wählen Sie `Library > Web Kit`. Das liegt nicht unter Cocoa. Es ist Teil eines separaten Frameworks, das Sie später mithilfe von Xcode in Ihr Projekt einbinden müssen. Dieses Framework enthält ein einzelnes visuelles Element namens `WebView`. In diesem Element wird später die Webseite gerendert. Klicken Sie das Icon an, das an Safari

erinnert, und ziehen Sie es in Ihr Fenster. Positionieren Sie den Web-View so, dass er den Rest des Fensters einnimmt. Ihr Fenster sollte jetzt etwa so aussehen:



Nehmen Sie sich etwas Zeit, um die Buttons, das Textfeld und den Web-View so anzuordnen, wie Sie es wünschen.

Man verliert bei der ganzen Klickerei und Zieherei schnell das große Ganze aus den Augen. In diesem Abschnitt haben Sie aus einer Palette von Elementen diejenigen ausgewählt, die Sie für Ihre Anwendung benötigen. Diese Elemente haben Sie dann im Hauptfenster so angeordnet, wie der Benutzer sie nach dem Start der Anwendung sehen soll.

Speichern Sie Ihre Arbeit mit ⌘ S oder `File > Save`. Schließen Sie das Library-Fenster. Für den Rest des Kapitels werden Sie es nicht mehr benötigen.

2.3 Das Interface mit dem Cocoa Simulator testen

Sie sollten ab und zu eine Pause einlegen und mit Ihrem Interface herumspielen, um sicherzugehen, dass es sich so verhält, wie Sie es wünschen. Zuerst müssen Sie den Interface Builder als Anwendung wählen (indem Sie irgendein IB-Fenster anklicken). Dann starten Sie den Cocoa Simulator über ⌘ R oder `File > Simulate Interface`.

Noch haben Sie keine Verknüpfungen vorgenommen, d. h. Sie können das Verhalten noch nicht testen. Im Moment können Sie nur prüfen, ob die Anwendung richtig aussieht. Verändern Sie die Größe des Fensters. Machen Sie es mal so richtig groß und so richtig klein.

Die Elemente verhalten sich wahrscheinlich nicht so, wie Sie es erwarten. Wenn Sie das Fenster sehr groß machen, erwarten Sie, dass der Web-View entsprechend mitwächst. Wenn Sie das Fenster in die Breite ziehen, sollen die Buttons und das Textfeld zusammenbleiben, und das Textfeld soll mitwachsen, um die Verschiebung auszugleichen.

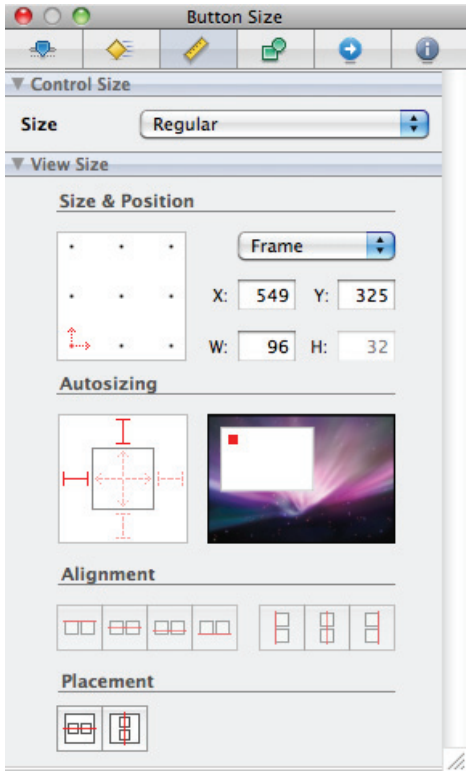
Das Problem ist, dass Sie Ihrer Anwendung noch nicht mitgeteilt haben, wie sich die Komponenten verhalten sollen, wenn das Fenster vergrößert oder verkleinert wird. Beenden Sie den Simulator mit ⌘Q oder `Cocoa Simulator > Quit Cocoa Simulator`.

Lassen Sie uns nun festlegen, wie sich die Größe der Komponenten ändert, wenn sich die Größe des Fensters ändert. Dazu müssen Sie im IB den *Size Inspector* öffnen. Das kann mit ⌘3 geschehen (Sie verwenden ⌘3, weil der Size Inspector der dritte Reiter des Inspector-Fensters ist). Sie können den Size Inspector auch mit der Maus über `Tools > Size Inspector` auswählen. Ist der Inspector offen, wählen Sie den dritten Reiter von links.[5] Das ist der, dessen Icon an ein Lineal erinnert. Sie können den Inspector über ⌘⇧I oder `Tools > Inspector` öffnen. Wenn Sie einen Button und dann den Size Inspector auswählen, sollten Sie so etwas sehen wie im Screenshot.

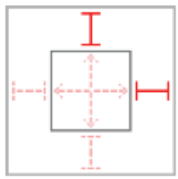
Um die Größeneinstellungen einer Komponente zu ändern, wählen Sie die Komponente aus und nehmen entsprechende Anpassungen im Size Inspector vor. Wir nehmen im Moment nur Änderungen an den „Autosizing"-Einstellungen vor. Sie sollten mit den Einstellungen der vier „Streben" herumexperimentieren, der I-förmigen Elemente an der Außenseite des inneren Quadrats.

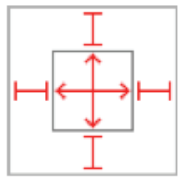
Während Sie sie ein- und ausschalten, zeigt die Animation auf der rechten Seite die jeweiligen Ergebnisse dieser Änderungen an. Auf diese Weise kontrollieren Sie, welche Seite der Komponente verankert wird, während das Fenster wächst oder schrumpft. Sie sollten auch die horizontalen und vertikalen „Federn" innerhalb des inneren Quadrats ausprobieren. Mit ihnen legen Sie fest, in welcher Richtung die Komponenten wachsen oder schrumpfen, wenn sich die Größe des Fensters verändert.

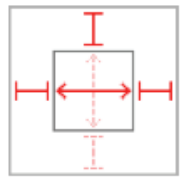
5 Es gibt viele Möglichkeiten, den jeweiligen Inspektor zu erreichen. Ich stelle mehr als eine vor, weil einige Leute lieber mit Menüs arbeiten, während andere Tastaturkürzel vorziehen. Wählen Sie die Variante, die Ihnen am meisten zusagt.



Hier sehen Sie die Einstellungen für beide Buttons, den Web-View und das Textfeld:


Buttons


Web View


Text Field

Wenn das Fenster wächst oder schrumpft, bleibt die Oberkante eines jeden Elements im gleichen Abstand von der Oberkante des Fensters. Die rechte Seite beider Buttons behält den gleichen Abstand zur rechten Seite des Fensters bei. Das Textfeld wächst horizontal, sodass seine beide Seiten den gleichen Abstand vom Rand des Fensters beibehalten. Der Web-View schließlich wächst horizontal *und* vertikal, um den Rest des Fensters aufzufüllen.

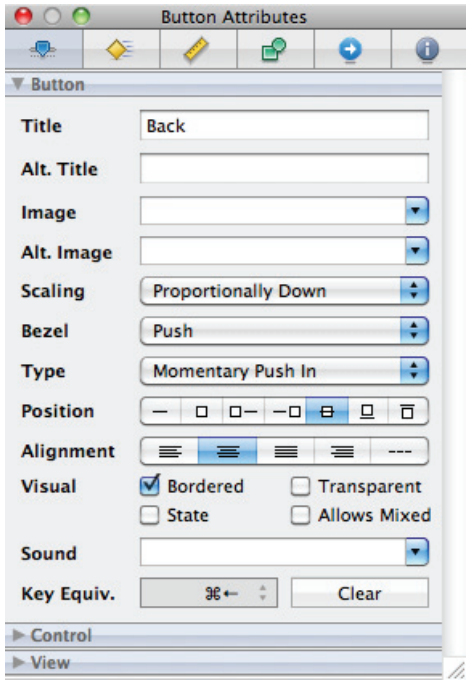
Speichern Sie Ihre Arbeit und testen Sie das Ergebnis mit dem Cocoa Simulator. Verkleinern Sie das Fenster weiter und vergrößern Sie es dann wieder. Wahrscheinlich wird Ihr Interface ab einer bestimmten Größe nicht mehr gut aussehen. Sie müssen die minimale Größe des Fensters festlegen. Zu diesem Zweck beenden Sie den Cocoa Simulator und wählen das `Window`-Objekt. Im Size Inspector sehen Sie einen Abschnitt mit dem Titel *Window Size*. Aktivieren Sie die Checkbox *Minimum Size* und klicken Sie den *Use Current*-Button an. Speichern Sie Ihre Arbeit. Nun ist der Benutzer nicht mehr in der Lage, das Fenster kleiner zu machen als diese Größe. Wenn Sie wollen, können Sie in gleicher Weise die maximale Größe des Fensters festlegen.

Testen Sie das Ergebnis erneut im Cocoa Simulator. Nehmen Sie sich ein wenig Zeit, um mit den Einstellungen für die Streben und Federn zu experimentieren und verschiedene minimale und maximale Größen für das Fenster auszuprobieren.

2.4 Das Interface fertigstellen

Die zwei Buttons tragen immer noch die Bezeichnung „Button“. Eine Möglichkeit, das Label eines Buttons zu ändern, besteht darin, ihn doppelt anzuklicken und den neuen Text einzugeben. Eine andere Möglichkeit ist die Wahl des *Attributes*-Reiters im Inspector. Das ist der Reiter ganz links, der mit dem symbolischen Schieberegler. Sie erreichen diesen Reiter auch über ⌘ I oder `Tools` > `Attributes Inspector`.

Geben Sie dem linken Button die Bezeichnung „Back“ („zurück“), indem Sie das Texteingabefeld rechts neben `Title` anklicken und dort *Back* eintragen. Wenn Sie einen anderen Bereich des Inspectors auswählen, sehen Sie, dass sich der Titel des Buttons ändert. Wo Sie gerade dabei sind, können Sie als Tastaturkürzel für diesen Button ⌘ ← festlegen. Dazu klicken Sie das graue Rechteck neben dem Label *Key Equiv.* an und geben den Linkspfeil (←) ein, während Sie die Befehlstaste (⌘) gedrückt halten.



Legen Sie als Titel des anderen Buttons „Forward“ („vor“) fest und als Tastaturkürzel ⌘ →. Als Nächstes wählen Sie `Window` und geben für den Titel des Fensters *SimpleBrowser* ein.

Schließlich wollen wir einen Standardwert vorgeben und dem Benutzer eine Erklärung anzeigen lassen, wie man den Browser benutzt. Setzen Sie den Titel des Textfelds auf *http://pragprog.com*, indem Sie es doppelt anklicken oder es auswählen und den Wert über den Attributes Inspector festlegen. Setzen Sie in der Drop-down-Liste für das Textfeld im Attributes Inspector auch den Wert für „Action“ auf „Send on Enter Only“. Damit wird das Textfeld so konfiguriert, dass es seinen Wert sendet, wenn der Benutzer die Enter-Taste drückt.

Welchen Wert sendet das Textfeld, wenn der Benutzer die Enter-Taste drückt? Das haben wir noch nicht festgelegt.

Sie haben nun alle visuellen Komponenten der Benutzerschnittstelle für den SimpleBrowser erzeugt. Nutzen Sie den Cocoa Simulator, um sich den Browser noch einmal anzusehen und die Größe des Fensters oder der Komponenten zu korrigieren. Der Browser kann immer noch nichts – aber das ändern wir im nächsten Abschnitt.

Den Nib lesen

Die bisher geleistete Arbeit wird als XML in der Datei `MainMenu.xib` gespeichert. Es gibt keinen Grund, den von Ihnen erzeugten XML-Code direkt zu lesen oder zu verändern. Dieses Format ist nur dazu gedacht, vom Interface Builder erzeugt und interpretiert zu werden. Sie können die Datei zum Spaß mit ihrem Lieblingseditor öffnen und sich ein wenig umsehen. Sie werden das Textfeld, die beiden Buttons und den Web-View finden. Das sieht nicht besonders hübsch aus, und Sie werden die GUI nicht in dieser Form aufbauen wollen, doch es kann beruhigend sein zu wissen, dass es sich um ein Format handelt, in dem man sich umsehen kann, wenn man will. Schließen Sie die Datei wieder, ohne irgendwelche Änderungen zu speichern. Wir werden uns `MainMenu.xib` am Ende des Kapitels noch einmal ansehen.

2.5 Die Komponenten verknüpfen

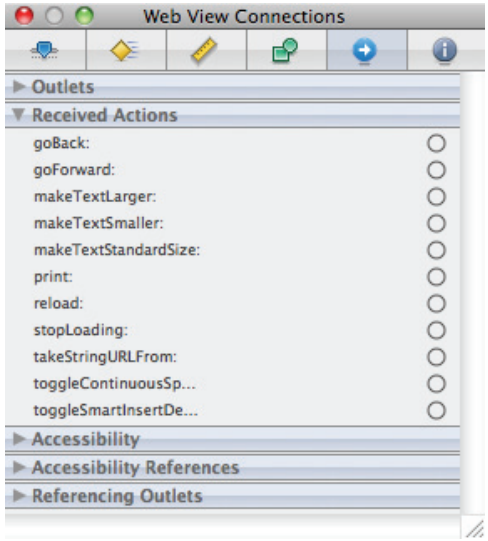
Sie haben den Interface Builder verwendet, um die visuellen Elemente ihrer Anwendung auszuwählen und zu konfigurieren. Sie haben sie im Fenster positioniert, ihr Erscheinungsbild geändert, die Standardwerte festgelegt, ihre Größe verändert und festgelegt, wie sie auf Größenänderungen des Fensters reagieren sollen. Nun sind wir bereit, die Elemente zu verknüpfen.

Beispielsweise wollen Sie den *Back*-Button mit dem Web-View verknüpfen. Wenn Sie den Button anklicken, soll der Web-View den Inhalt der vorherigen URL anzeigen. Der Durchschnitts-Button weiß nichts über Web-Views. Ein Web-View sollte hingegen wissen, wie er sich in seiner History vor- und zurückbewegt, und sollte über eine externe Quelle angestoßen werden können. Sie müssen also nur die Fähigkeit des Web-View, sich in seiner History zurückzubewegen, mit dem Klick auf den Button verknüpfen.

Ebenso weiß ein Textfeld nichts über URLs und Web-Views. Müsste ein Textfeld alles über den Text wissen, der eingegeben werden könnte, wäre die API riesig und instabil. Doch ein Web-View sollte wissen, wie man einen URL aus einem String eines anderen Elements extrahiert. Wir müssen also nur die „Extrahiere URL aus String"-Fähigkeit des Web-View mit der Komponente verknüpfen, die den String zur Verfügung stellt.

Den Web-View verknüpfen

Es funktioniert nicht immer, aber es kann hilfreich sein, daran zu denken, wer was weiß. In diesem Fall weiß der Web-View das meiste. Ein Großteil dessen, was ein Objekt tun kann, wird im Interface Builder über den Connections Inspector sichtbar.



Um sich einen Teil dessen anzusehen, was man mit dem Web-View machen kann, wählen Sie den Web-View im Layoutfenster und öffnen den Connections Inspector. Ist der Web-View ausgewählt und bereits der Inspector geöffnet, können Sie den blauen Rechtspfeil anklicken, um den Reiter des Connections Inspectors zu aktivieren. Sie können auch ⌘5 drücken oder `Tools > Connections Inspector` verwenden. Die folgende Abbildung zeigt die Aktionen, die der Web-View empfangen kann. Das sind die einzigen, die wir im Moment benötigen.

Der nächste Schritt hat für mich etwas von Zauberei.

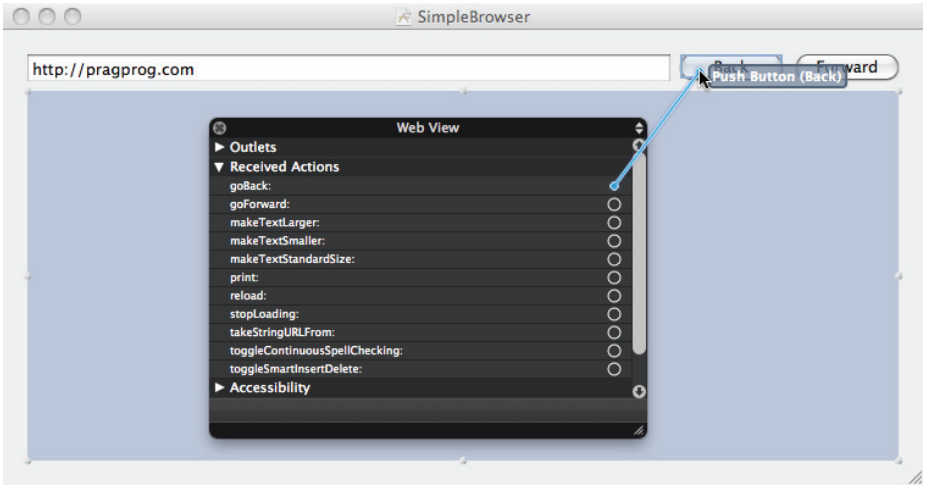
Klicken Sie im Connections Inspector in den Kreis rechts neben `goBack:` und halten Sie die Maustaste gedrückt. Ziehen Sie den Mauszeiger über den von Ihnen angelegten *Back*-Button. Sobald Sie mit dem Ziehen beginnen, erscheint eine blaue Linie. Wenn Sie die Maus über den *Back*-Button bewegen, erscheint eine graue Box mit den Worten *Push Button (Back)*. Lassen Sie die Maustaste los, und die Verbindung ist hergestellt.

Hier noch einmal die Wiederholung in Zeitlupe. Ein `Web-View` weiß, wie man sich in der Browser-History zurückbewegt. Der Button weiß, wie er eine Aktion initiiert, wenn er gedrückt wird. Wir haben den Connections

Inspector genutzt, um die beiden miteinander zu verbinden. Nachdem wir die Verbindung hergestellt haben, wird bei jedem Klick auf den *Back*-Button die Web-View-Methode `goBack:` aufgerufen. Und wenn es eine vorherige Seite in der Browser-History gibt, wird die entsprechend geladen.

Sehen Sie sich noch mal den Connections Inspector an. Der von Ihnen angeklickte Kreis neben `goBack:` ist nun ausgefüllt. Außerdem ist `goBack:` jetzt auch visuell mit `Push Button (Back)` verbunden. Sehen Sie das *x* links neben `Push Button (Back)`? Klicken Sie darauf, und Sie haben die Verbindung wieder aufgehoben.

Sie können `goBack:` und den *Back*-Button wie eben wieder verbinden, oder Sie klicken den Web-View mit der rechten Maustaste oder einem Control-Klick (Strg-Klick) an, um ein Heads-up-Display zu öffnen. Wie zuvor klicken Sie den Kreis neben `goBack:` an und ziehen die Maus über den *Back*-Button. Lassen Sie die Maustaste los, und Sie haben `goBack:` wieder mit dem *Back*-Button verbunden.

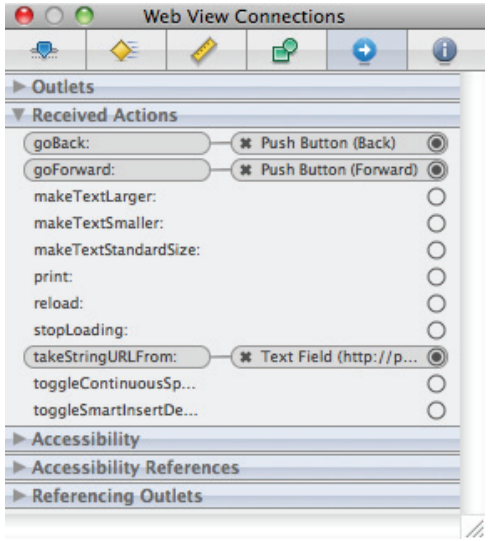


Wir müssen noch zwei weitere Verbindungen herstellen. Wählen Sie erneut den `Web-View` aus und nutzen Sie den Connections Inspector oder das Heads-up-Display.

1. Verknüpfen Sie die Aktion `goForward:` mit dem *Forward*-Button. Das ermöglicht es dem Benutzer, den *Forward*-Button anzuklicken, um sich in der Browser-History vorwärtszubewegen.

2. Verknüpfen Sie die Aktion `takeStringURLFrom:` mit dem Textfeld. So kann der Benutzer eine URL in das Textfeld eingeben, um eine Webseite zu laden.

Wenn Sie damit fertig sind, sollte der Connections Inspector wie folgt aussehen:



Als ob Sie nicht schon genug Optionen hätten, können Sie diese Verbindungen auch in entgegengesetzter Richtung herstellen. Wählen Sie beispielsweise den *Back*-Button aus und sehen Sie sich ihn im Connections Inspector an. Im Abschnitt *Sent Actions* sehen sie einen einzelnen Eintrag, `selector:`[6]. Wählen Sie ihn aus, ziehen Sie ihn in den WebView und lassen Sie die Maustaste los. Eine graue Box mit allen WebView-Aktionen erscheint.

Wählen Sie `goBack:` aus. Sie haben nun die Verbindung in umgekehrter Richtung hergestellt. Sie werden schnell ein Gefühl für die von Ihnen bevorzugte Methode und Richtung entwickeln. Suchen Sie sich aus, was Ihnen am meisten liegt.

Den Browser im Simulator testen

Speichern Sie Ihre Arbeit ab und öffnen Sie den Cocoa Simulator. Klicken Sie das Textfeld an und drücken Sie ↵. Die Pragmatic Programmer-Homepage erscheint. Geben Sie eine weitere URL ein und drücken Sie ↵. Auch wenn es größtenteils funktioniert, möchte ich ein paar Anmerkungen machen:

6 Ein Selektor ist der Name der aufgerufenen Methode, und der Doppelpunkt deutet an, dass diese Methode ein einzelnes Argument verlangt. In Kapitel 3, *Methoden und Parameter*, werden Sie noch viel mehr über Selektoren lernen.

- Vergessen Sie nicht, das `http://` anzugeben. Es ist ein sehr einfacher Webbrowser.
- Wählen Sie weder übermäßig komplizierten Seiten noch solche mit Authentifizierung. Es ist ein *sehr* einfacher Webbrowser.[7]
- Sobald eine zweite Seite vorhanden ist, können Sie die *Back*- und *Forward*-Buttons anklicken, um sich zwischen den beiden Seiten vor- und zurückzubewegen.

Hervorragend! Zumindest im Simulator besitzen wir einen funktionierenden Webbrowser.

Unser Browser verfügt nicht einmal über all die Features, die Sie selbst vom einfachsten Browser erwarten würden. Zum Beispiel erhalten Sie beim Laden einer Seite keinerlei Feedback. Wenn Sie SimpleBrowser das erste Mal ausführen, könnten Sie glauben, dass gar nichts passiert, bis nach einer kurzen Pause die Seite erscheint. Auch die URL ändert sich nicht, wenn Sie *Back* und *Forward* anklicken. Wenn Sie zuerst *http://pragprog.com* und dann *http://apple.com* laden und dann den *Back*-Button anklicken, erscheint die Pragmatic Programmers-Homepage, doch die URL zeigt immer noch *http://apple.com* an. Wir werden uns um diese Geschichten in den nächsten Kapiteln kümmern.

2.6 Den Build korrigieren

Man könnte meinen, wir wären fertig: Man kann mit der Anwendung im Cocoa Simulator herumspielen. Ihre Arbeiten im Interface Builder sind korrekt und vollständig, und Sie mussten keinerlei Code für dieses SimpleBrowser-Projekt schreiben. Allerdings besitzen Sie noch keine Anwendung, die Sie weitergeben könnten. Sie müssen zurück zu Xcode und ein Release erzeugen.

In Xcode klicken Sie auf *Build & Run*. Der Build ist erfolgreich und es sieht so aus, als würde SimpleBrowser mit dem Laden beginnen. Das Icon hüpft eine Weile im Dock auf und ab, aber nichts passiert. Abhängig von Ihrem Setup finden Sie sich mit einem Stacktrace oder einer Warnung im Debugger wieder.

7 Es geht in diesem Beispiel nicht darum, einen robusten Webbrowser zu entwickeln. Vielmehr verwenden wir den Browser als Beispielanwendung, die uns die Einführung in die Cocoa-Programmierung erleichtert. Momentan können Sie beispielsweise keine Seiten laden, die Flash enthalten.

Arrggghhhh! Klicken Sie auf den *Stop*-Button oder wählen Sie `Run > Stop`.[8] Wir müssen noch das `WebKit.framework` einbinden.

Ich hatte schon erwähnt, dass wir es einbinden müssen. Wir hätten das tun können, als wir den Web-View aus der Library in unser Fenster zogen. Die Chancen stehen gut, dass Sie sich nun daran erinnern, nachdem Sie gesehen haben, welche Probleme auftreten können. Im *Joe fragt ...* auf dieser Seite finden Sie eine Erläuterung, wie Sie das Problem selbst hätten diagnostizieren können.

Joe fragt...

Wie hätte ich erkennen können, dass ich das WebKit-Framework einbinden muss?

Zuerst öffnen Sie den Debugger über Run > Debugger oder ⇧⌘Y. Je nachdem, wie Ihr Xcode konfiguriert ist, sehen Sie auch ein Icon (das an Insektenspray erinnert), das Sie anklicken können. In der linken oberen Ecke Ihres Xcode-Fensters sehen Sie die Warnung, dass die Anwendung aufgrund einer „uncaught Exception" beendet wurde:

```
TERMINATING_DUE_TO_UNCAUGHT_EXCEPTION
```

Der Stacktrace ist nicht besonders hilfreich, doch wenn Sie auf die Konsole schauen, wird das Problem schnell deutlich. Öffnen Sie die Konsole über `DBR` oder `Run > Console`. Sie sehen das `gdb`-Prompt. Scrollen Sie über das Stacklisting zurück nach oben. Direkt darüber sehen Sie eine Reihe von Zeilen, die alle mit einem Timestamp, dem Wort *SimpleBrowser* und eckigen Klammern um einen Identifier (der die Prozessnummer enthält) beginnen. Sobald Sie diese Informationen ausblenden, sollte so etwas übrig bleiben:

```
An uncaught exception was raised
-[NSKeyedUnarchiver decodeObjectForKey:]:
                    cannot decode object of class (WebView)
Terminating app due to uncaught exception
                    'NSInvalidUnarchiveOperationException'
```

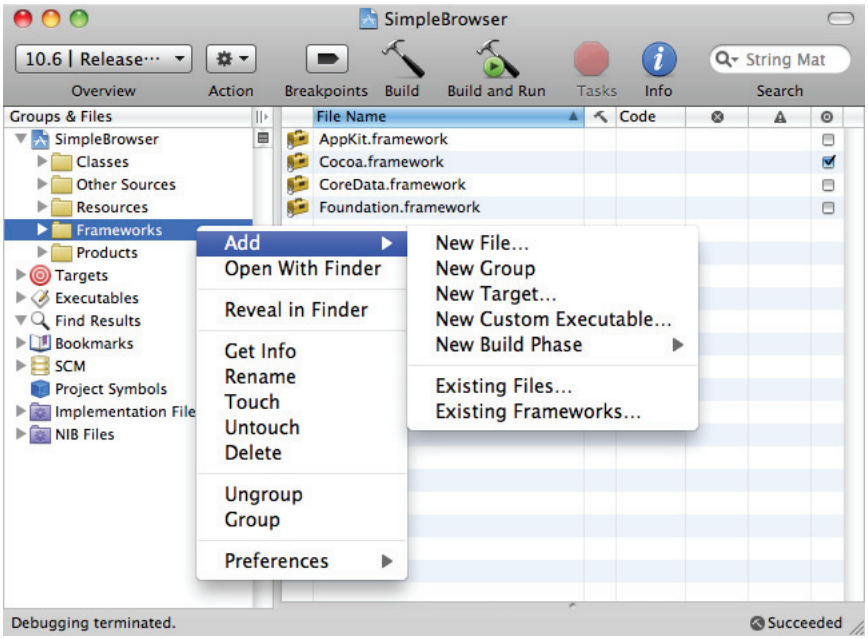
Aha! Das Problem hat etwas mit der „Dearchivierung" eines `WebView`-Objekts zu tun. Das Nib ist im Wesentlichen ein Archiv von Objekten und ihren Verbindungen. Sie müssen das `WebKit`-Framework laden, damit der Web-View erfolgreich wiederhergestellt werden kann.

8 Wenn Ihre Xcode-Installation nicht so aussieht wie meine, können Sie die Menüleiste jederzeit über `View > Customize Toolbar...` anpassen. Außerdem können Sie über `Xcode > Preferences` viele weitere Anpassungen vornehmen.

Als wir die `WebView`-Instanz aus der Library übernahmen, haben wir gesehen, dass sie Teil des `WebKit`-Frameworks, nicht aber des `Cocoa`-Frameworks ist. Ich hätte erwartet, dass Xcode automatisch das `WebKit`-Framework in das entsprechende Projekt einbindet, aber das ist nicht der Fall: Sie müssen es selbst machen.

In der Projektansicht wählen Sie den `Frameworks`-Ordner in der SimpleBrowser-Gruppe aus. Sie sollten vier Frameworks sehen, die bereits in das Projekt eingebunden sind. Ein Blick auf die rechte Spalte zeigt aber, dass für unser Target nur das `Cocoa.framework` aktiv ist.

Klicken Sie den `Frameworks`-Ordner mit der rechten Maustaste oder mit einem Control-Klick an und wählen Sie `Add > Existing Frameworks...`.



Wählen Sie das `WebKit.framework` und stellen Sie sicher, dass nach dem Einbinden die *Target*-Checkbox aktiviert wird.[9]

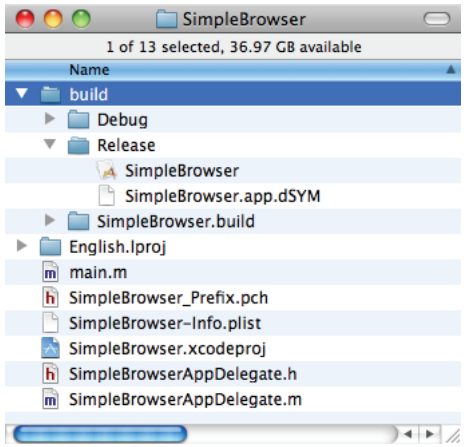
Herzlichen Glückwunsch! Sie können Ihre Anwendung nun unter Xcode kompilieren und ausführen.

9 Sie können ein Framework auch einbinden, indem Sie ein Ziel (Target) mit `Add > Existing Frameworks ...` einfügen, oder im *General*-Reiter, das erscheint, sobald Sie *Get Info* wählen.

2.7 Ihren Browser weitergeben

Nachdem Sie nun einen funktionierenden Webbrowser entwickelt haben, wollen Sie Ihre Anwendung mit Ihren Freunden teilen – zumindest mit denjenigen, die Mac OS X nutzen. Stellen Sie sicher, dass Ihre Einstellung für Active Build Target mit Release und nicht mit Debug eingestellt ist. Sie erreichen das über `Project > Set Active Build Configuration > Release` oder über das Drop-down-Menü *Active Build Configuration* in Xcode, wenn es entsprechend Ihren Xcode-Einstellungen sichtbar ist.

Sobald das Build-Target gesetzt ist, klicken Sie erneut auf *Build & Run*. Diesmal sollte die SimpleBrowser-Anwendung korrekt kompiliert, gelinkt und gestartet werden. Werfen Sie einen Blick in das `build/Release`-Verzeichnis des Projekts.



Sie sollten die SimpleBrowser-Anwendung sehen. Da wir *Release* und nicht *Debug* als Target gewählt haben, kann diese Datei weitergegeben und auf anderen Maschinen ausgeführt werden. Die einfachste Möglichkeit besteht darin, die Datei `SimpleBrowser` innerhalb des `build/Release`-Verzeichnisses mit der rechten Maustaste anzuklicken und dann *Compress „SimpleBrowser“* zu wählen. Das erzeugt die Datei `SimpleBrowser.zip`, die Sie per Mail verschicken können.

2.8 Übung: Und jetzt das Ganze noch mal von vorne

In diesem Kapitel haben Sie einen funktionsfähigen Webbrowser entwickelt, ohne eine einzige Zeile Code zu schreiben. Sie haben Apples `WebKit`-Framework die eigentliche Arbeit erledigen lassen. Sie haben den Großteil Ihrer Zeit damit verbracht, die Anwendung korrekt aussehen

zu lassen, und mit ein paar Klicks und ein wenig Drag-and-Drop gegen Ende haben Sie das gewünschte Verhalten erzeugt.

Sie haben gelernt, wie man diese Arbeiten in einzelnen Schritten erledigt, doch Sie werden normalerweise nicht auf diese Weise vorgehen. Wenn Sie das noch mal ganz von vorne machen müssten, würden Sie einzelne Schritte kombinieren. Wenn Sie den ersten Button an seine Position ziehen, würden Sie ihn wahrscheinlich gleich in „Back“ umbenennen und mit dem Web-View verknüpfen.

Auch wenn Sie versucht sein werden, diese Übung zu überspringen: Sie werden doch sehr viel damit gewinnen, wenn Sie ganz von vorne beginnen und den Browser noch einmal entwickeln. Da Ihnen niemand Schritt für Schritt erklärt, was als Nächstes zu tun ist, werden die Teile langsam anfangen, sich ineinander zu fügen. Nehmen Sie sich fünf Minuten Zeit, um einen eigenen Webbrowser zu entwickeln, und Sie werden Ihr Wissen festigen.

2.9 Die .nib-Datei

Alle Arbeiten, die Sie in diesem Kapitel durchgeführt haben, werden in einer *.nib*-Datei festgehalten. Ein *Nib* ist ein Archiv mit Objekten. Bei Cocoa enthalten *.nib*-Dateien alle Informationen, die Sie benötigen, um Ihre UI-Elemente zur Laufzeit zum Leben zu erwecken. Eine typische Cocoa-Anwendung besitzt viele *.nib*-Dateien, die nur geladen werden, wenn sie benötigt werden, um Instanzen der Objekte zu erzeugen, aus denen die Benutzerschnittstelle besteht. Das gilt aber auch für nichtvisuelle Objekte, wie Sie in Kapitel 8, *Einen Controller erzeugen*, noch sehen werden.

Die *.nib*-Datei stellt einen Objektgraphen der Objekte dar, die Sie im Interface Builder eingesetzt haben. Ihre Benutzerschnittstelle und andere im Nib festgehaltene Objekte können auf diese Weise beim Start der Anwendung wieder aufgebaut werden. Sie definieren Ihre Klassen in Xcode und instanziieren Klassen unter Xcode oder Interface Builder in Objekte. Eine mit dem Interface Builder aufgebaute *.nib*-Datei ist ein schockgefrorener Graph von Objekten, der zur Laufzeit wieder zum Leben erweckt wird.

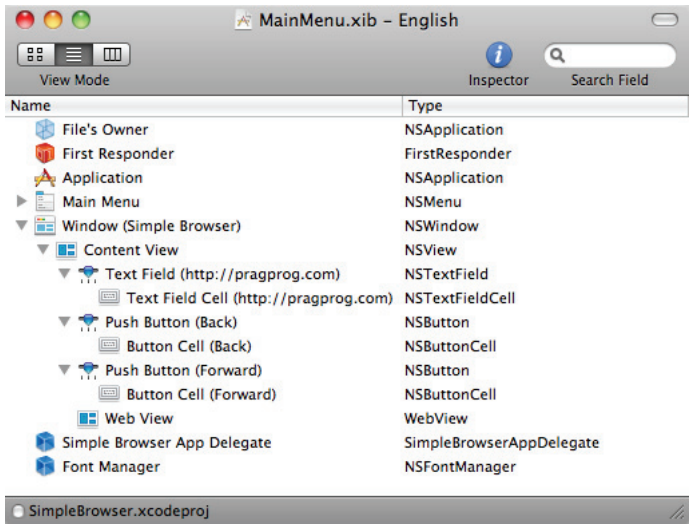
Wir werden im Verlauf dieses Buches immer wieder zum Interface Builder und zu *.nib*-Dateien zurückkehren. Umfassende Informationen finden Sie außerdem in Apples *Interface Builder User Guide* [App08e]. Scott Stevenson bietet ebenfalls einen schönen Schnelleinstieg an, in

dem eine andere Anwendung ohne Programmcode entwickelt wird.[10] Er verwendet noch Xcode 3.1, unter Snow Leopard sieht das Ganze also etwas anders aus.

Das Dokumentenfenster

Lassen Sie uns einen Blick auf die Objekte im `MainMenu`-Nib werfen. Klicken Sie `MainMenu.xib` doppelt an, um es erneut im Interface Builder zu öffnen. Diesmal öffnen Sie das Dokumentenfenster (Document window) über `Window > Document` oder das Tastaturkürzel ⌘ 0.

Ich bevorzuge die Listenansicht, die man über den mittleren der drei Reiter in der linken oberen Ecke aktivieren kann. Ich habe auch die Dreiecke für das `Window`-Objekt und die darin enthaltenen Objekte geöffnet. Diese Hierarchie umfasst alle GUI-Komponenten des Layoutfensters.



Wenn Sie sich die `Window`-Hierarchie ansehen, erkennen Sie, dass das Fenster einen Content View besitzt, der die vier Komponenten enthält, die wir in das Fenster gezogen haben. Die *.nib*-Datei enthält auch das Hauptmenü und eine Reihe unsichtbarer Elemente. Ich will jetzt nicht darauf eingehen, was die einzelnen Elemente genau machen, ich wollte Ihnen aber diese Ansicht der Objekte im Nib zeigen, weil wir sie später benutzen werden, um visuelle und unsichtbare Elemente miteinander zu verknüpfen. Dieses Fenster hilft dabei, im Interface Builder aufgebaute Dinge mit dem unter Xcode entwickelten Code zu verknüpfen. Sie

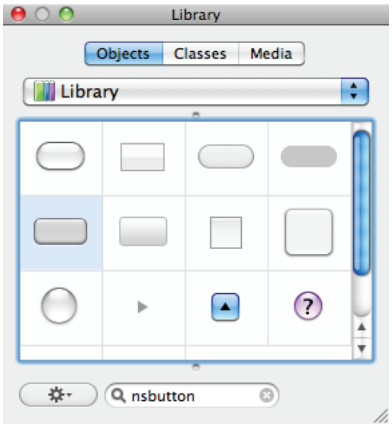
10 http://cocoadevcentral.com/d/learn_cocoa/

bietet auch eine bequeme Möglichkeit, um Komponenten auszuwählen, die im Layoutfenster verschachtelt sind. Sollten Sie das Layoutfenster aus den Augen verlieren, kommt es durch einen Doppelklick auf das Window-Element (SimpleBrowser) im Dokumentenfenster wieder nach vorn.

Die XML-Darstellung der .nib-Datei

Das Dokumentenfenster liefert uns eine Ansicht der *.nib*-Datei. Einen anderen Blickwinkel bietet uns der XML-Code, der während der Entwicklungsphase im Nib gespeichert wird. Sehen wir uns noch einmal die in diesem Kapitel aufgebaute und verwendete *.nib*-Datei an.

Unsere *.nib*-Datei enthält zwei Buttons, die Instanzen der Klasse `NSButton` sind. Wenn Sie in der Interface Builder Library nach *NSButton* suchen, werden Sie nicht nur einen Button finden, sondern mehr als ein Dutzend:



Wenn Sie einen `NSButton` aus der Library in den View ziehen, erzeugen Sie eine Instanz der Klasse `NSButton`. Sie können jeden `NSButton` aus der obigen Abbildung nehmen und in die von Ihnen gewünschte Instanz von `NSButton` verwandeln. Doch Apple stellt Ihnen eine Palette von Buttons zur Verfügung, die sehr unterschiedlich aussehen. Während Sie sich in der Library umsehen, geht es also nicht so sehr darum, einfach einen `NSButton` zu finden. Vielmehr suchen Sie nach einem Button, der ein bestimmtes Aussehen hat oder in einer bestimmten Art und Weise verwendet wird. Das heißt, Sie wählen die Komponenten aus Sicht des Endanwenders aus.

Sie wählen den Button aus, platzieren und konfigurieren ihn. Zumindest einen Teil der abgespeicherten Informationen können Sie sehen, wenn Sie sich die XML-Datei ansehen.

Wir wollen `MainMenu.xib` mit einem Texteditor öffnen, um uns ein wenig umzusehen. Sie werden sich eine *.nib*-Datei nie wieder in dieser Form ansehen müssen. Außerdem müssen Sie mir versprechen, niemals etwas an einer *.nib*-Datei zu verändern, wenn Sie sie mit einem Texteditor geöffnet haben. Der Interface Builder ist das einzige Tool, mit dem Sie Nibs anlegen, betrachten und modifizieren sollten.

Wenn Sie es nicht schon getan haben, öffnen Sie `MainMenu.xib` mit einem Texteditor. Suchen Sie nach dem Text *NSButton* und suchen Sie nach dem Teil der Datei, der das Aussehen ihres *Back*-Buttons beschreibt:

UsingWhatsThere/SampleNib/MainMenu.xib

```
<object class="NSButton" id="164086064">
        <reference key="NSNextResponder" ref="439893737"/>
        <int key="NSvFlags">265</int>
        <string key="NSFrame">{{277, 258}, {93, 32}}</string>
        <reference key="NSSuperview" ref="439893737"/>
        <bool key="NSEnabled">YES</bool>
        <object class="NSButtonCell" key="NSCell" id="941085700">
                <int key="NSCellFlags">67239424</int>
                <int key="NSCellFlags2">134217728</int>
                <string key="NSContents">Back</string>
                <reference key="NSSupport" ref="640083843"/>
                <reference key="NSControlView" ref="164086064"/>
                <int key="NSButtonFlags">-2038284033</int>
                <int key="NSButtonFlags2">268435585</int>
                <string key="NSAlternateContents"/>
                <string type="base64-UTF8" key="NSKeyEquivalent">75yCA</string>
                <int key="NSPeriodicDelay">200</int>
                <int key="NSPeriodicInterval">25</int>
        </object>
</object>
```

Das beschreibt ein Objekt vom Typ `NSButton`. Die `x`- und `y`-Koordinaten sowie die Höhe und Breite finden Sie in der Zeile, die `NSFrame` als Schlüssel (`key`) verwendet. Das `NSButton`-Objekt enthält außerdem ein Objekt vom Typ `NSButtonCell`. Wir haben uns noch nicht über `NSButtonCells` unterhalten, aber Sie können erkennen, dass dort der Name des Buttons und das Tastaturkürzel festgelegt werden.

Die *.nib*-Datei enthält darüber hinaus Objekte, die die von Ihnen hergestellten Verbindungen repräsentieren. Suchen Sie beispielsweise nach dem String `takeStringURLFrom:`. Sie sollten Folgendes in `MainMenu.xib` vorfinden:

UsingWhatsThere/SampleNib/MainMenu.xib

```
<object class="IBConnectionRecord">
          <object class="IBActionConnection" key="connection">
                    <string key="label">takeStringURLFrom:</string>
                    <reference key="source" ref="1029174864"/>
                    <reference key="destination" ref="109215417"/>
          </object>
          <int key="connectionID">459</int>
</object>
```

Sie können nach den Referenznummern für das Ziel (destination) und die Quelle (source) suchen und werden erkennen, dass dieses Fragment die Verbindung zwischen dem `WebView`-Objekt mit dem Label `takeStringURLFrom:` und dem Zielobjekt `NSTextField`-Objekt beschreibt. Sie werden gleich erfahren, wie man das etwas anders ausdrücken kann. Wie Sie im nächsten Kapitel sehen werden, bedeutet das einfach, dass das Ziel (das `WebView`-Objekt) die Nachricht `takeStringURLFrom:` erhält. Dabei wird das `NSTextField`-Objekt als Sender übergeben.

In unserem Fall ist der Lebenszyklus des *.nib*-Objekts recht einfach: Beim Start der Anwendung wird der Objekt-Graph in den Speicher geladen und dearchiviert. Die Komponenten werden initialisiert. Dann werden alle Verbindungen zwischen den Objekten hergestellt, und das Hauptmenü erscheint.

Natürlich benötigt jede sinnvolle Anwendung auch ein wenig Programmcode – und das ist der nächste Halt auf unserer Tour.

Kapitel 3

Methoden und Parameter

Nun haben Sie also einen funktionierenden Webbrowser entwickelt. Einerseits sind Sie stolz auf sich: Sie haben mit ein wenig Drag-and-Drop hier und ein paar Verknüpfungen da einen Browser aufgebaut. Starke Sache.

Andererseits fühlt sich das nicht nach richtigem Programmieren an. Ein *richtiger* Programmierer würde nicht mit einer Reihe von GUI-Tools arbeiten. Ein richtiger Programmierer würde die Objekte mit bloßen Händen bauen. Er würde Ächzen und männliche Laute von sich geben, während er mit dem Speichermanagement kämpft. Haben Sie jetzt den Eindruck, ein Weichei zu sein? Hmm, und Sie haben noch nicht mal ein eigenes „Hallo, Welt!“-Programm geschrieben!

Nun, Ihr „Hallo, Welt!“-Programm muss noch bis zum nächsten Kapitel warten. Bevor Sie anfangen, eigene Objekte zu entwickeln und eigene Methoden zu schreiben, müssen Sie sich zunächst damit vertraut machen, wie Objective-C einem Ziel eine Aktion sendet und wie man durch die API-Dokumentation navigiert, um herauszufinden, welche Nachrichten man den Objekten in den Cocoa-APIs senden kann. Dieses Kapitel ist Ihre Einführung in Nachrichten unter Objective-C und Cocoa.

3.1 Nachrichten ohne Argumente senden

Im Abschnitt 2.5, *Die Komponenten verknüpfen*, auf Seite 23 haben Sie den Interface Builder genutzt, um eine Verbindung von Ihrem *Back*-Button zur Aktion `goBack:` des Web-View herzustellen. Bei jedem Klick auf den *Back*-Button wird also die *WebView*-Methode `goBack` aufgerufen.[1]

Lassen Sie uns annehmen, dass wir die beiden an dieser Aktion beteiligten Objekte instanziiert haben. Wir besitzen eine Instanz von `NSButton` namens `backButton` und eine Instanz von `WebView` namens `myWebView`. Wenn der Benutzer den *Back*-Button anklickt, wird im Grunde die folgende Nachricht gesendet:

```
[myWebView goBack]
```

Die eckigen Klammern und alles darin bilden den *Nachrichtenausdruck (message expression)*. In diesem einfachsten aller Fälle besteht er aus nur zwei Teilen: dem Empfänger und der Nachricht. Das `myWebView`-Objekt ist der Empfänger. Er bildet das Ziel (*target*) – das Objekt, an das Sie die Nachricht senden. In unserem Beispiel heißt diese Nachricht `goBack`.

Wir senden eine Nachricht ohne Argumente. In dieser Form sieht der Aufruf wie folgt aus:

```
[empfänger nachricht]
```

Je nachdem, welche Programmiersprache Sie gerade benutzen, sieht das für Sie wie ein Funktions- oder wie ein Methodenaufruf aus. Mit anderen Worten sieht die Objective-C-Nachricht

```
[myWebView goBack]
```

in Ihrer Programmiersprache etwa so aus:

```
myWebView.goBack()
```

Wie Sie es wohl erwarten, können Sie Nachrichten miteinander verbinden, genau wie es bei Methoden möglich ist. In Java oder C# können Sie Folgendes schreiben:

```
myWebView.ersteMethode().zweiteMethode()
```

Das wendet `ersteMethode()` auf `myWebView` an und dann `zweiteMethode()` auf das Ergebnis.

---

1 Vielleicht ist Ihnen aufgefallen, dass ich die Sache ein wenig vereinfacht habe, indem ich das `:` am Ende von `goBack` weggelassen habe. Ich werde das gleich wieder ändern, aber es macht uns das Ganze für den Anfang etwas einfacher.

In Objective-C können Sie den gleichen fiktiven Code so schreiben:

```
[[myWebView ersteMethode] zweiteMethode]
```

Man liest diese verschachtelten Aufrufe von innen nach außen. Zuerst wird die Nachricht `ersteMethode` an `myWebView` gesendet. Das Ergebnis dieses Aufrufs ist dann das Ziel der Nachricht `zweiteMethode`.

Welche Nachrichten können Sie nun eigentlich an `myWebView` senden? Sie werden viel Zeit damit verbringen, sich die Dokumentation der von Apple bereitgestellten APIs anzusehen. Werfen wir also einen Blick auf die Dokumentation zu `WebView` und die Beschreibung dieser `goBack`-Methode.

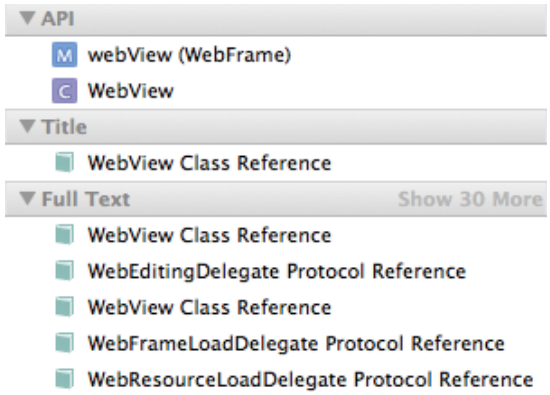
3.2 Die Dokumentation lesen

Apple bietet eine umfassende Dokumentation an, die Ihnen dabei hilft herauszufinden, was Sie bei der Entwicklung von Cocoa-Anwendungen verwenden können und wie Sie es nutzen. Sie können in Xcode über `Help > Developer Documentation` auf die Dokumentation zugreifen. Natürlich werden Sie sich auch ohne meine Hilfe in der Dokumentation zurechtfinden – ich möchte nur ein paar Dinge hervorheben, die Ihnen die Arbeit ein wenig erleichtern können.

Geben Sie *webview* in das Suchfeld in der oberen rechten Ecke des Doc-Viewers ein und experimentieren Sie mit den verschiedenen Ergebnissen, die Sie erhalten, je nachdem, ob Sie die Dokumentation für das iPhone, Mac OS X, Xcode oder eine beliebige Kombination davon durchsuchen.[2] Sie können auch wählen, ob Sie an Ergebnissen interessiert sind, die mit dem Suchstring anfangen, darin enthalten sind oder exakt übereinstimmen.

Sie können die Suche auf unterschiedliche Weise verfeinern. Im Moment beschränken wir die Dokumentation (Doc Set) auf die Mac OS X 10.6 Core Library, aktivieren alle Sprachen und wählen *Exact* (für eine genaue Übereinstimmung). Die Ergebnisse werden unter den Überschriften API, Title und Full Text zusammengefasst:

2 Die Dokumentation sowie die Art und Weise, wie man sie durchsucht und die Ergebnisse filtert, hat sich in Snow Leopard geändert. Die hier beschriebenen Anweisungen gelten für Xcode 3.2 (und höher), das mit Snow Leopard veröffentlicht wurde. Wenn Sie mit Leopard (oder früher) arbeiten, sollten die Unterschiede auf der Hand liegen.

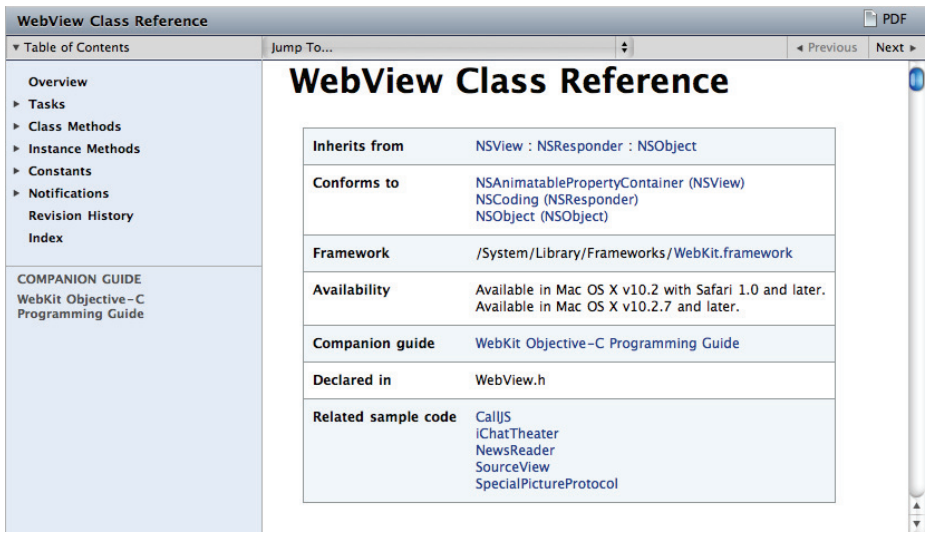


Die ersten beiden Ergebnisse unter API sind die `webView`-Methode aus der `Web-Frame`-Klasse sowie die `WebView`-Klasse. Der einzige Eintrag unter *Title* und der erste Eintrag unter *Full Text* verweisen ebenfalls auf die Dokumentation der `WebView`-Klasse.

Wir erhalten unter *Full Text* mehr Ergebnisse, weil *webview* nur im Titel eines Dokuments auftaucht, aber im Text von mehr als 30 Dokumenten vorkommt.

Wenn Sie die Dokumentation auf das iPhone ausdehnen und *Contains* anstelle von *Exact* auswählen, erscheint in diesen Kategorien auch das iPhone-Gegenstück zu Cocoas `WebView`, die Klasse `UIWebView`.

Wählen Sie die Zeile mit der `WebView`-Klasse aus und Sie sehen die Klassenreferenz für `WebView`. Hier der Anfang des Listings:



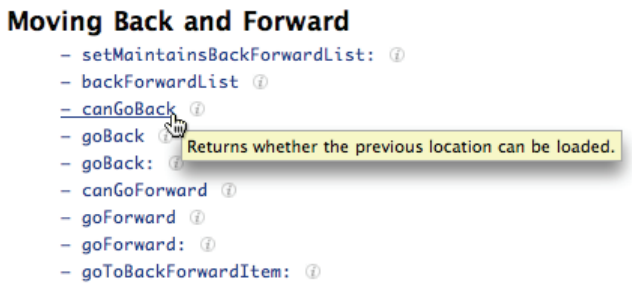
Auf der rechten Seite können Sie die Vererbung von `WebView` bis zur Cocoa-Rootklasse `NSObject`. zurückverfolgen. `WebView` erweitert `NSView`, die wiederum `NSResponder` erweitert, und die erweitert schließlich die Rootklasse `NSObject`.

Sie können außerdem erkennen, welches Framework Sie einbinden müssen, um diese Klasse nutzen zu können. Objective-C verwendet Header-Dateien[3] – in diesem Fall ist `WebView` in der Datei `WebView.h` deklariert, die Teil des `WebKit`-Frameworks ist. Der Abschnitt *Availability* (Verfügbarkeit) in der Methodenbeschreibung gibt an, ob eine Methode auch verfügbar ist, wenn man für ältere Versionen von Mac OS X entwickeln möchte.

Sie können das Dokument nach bestimmten Methodennamen durchsuchen, aber es gibt drei Links in der grauen Box auf der linken Seite, die Ihnen bei der Nutzung einer Cocoa-Klasse helfen.

- *Overview*: Eine kurze Zusammenfassung zur Aufgabe der Klasse und dazu, wie man sie nutzt. Enthält Verweise auf spezifische Methoden, die aufgerufen werden müssen, sowie auf Klassen, die mit der beschriebenen Klasse häufig zusammenarbeiten.

- *Tasks*: Alphabetische Listings der Klassen- und Instanzmethoden finden Sie an anderer Stelle in der Dokumentation, aber hier finden Sie eine Zusammenfassung der Methoden nach Funktion, d. h. welche Aufgaben Sie mit ihnen erledigen können. Der nachfolgende Screenshot zeigt beispielsweise die Methoden, mit denen man sich vor- und zurückbewegen kann. Ich habe den Mauszeiger über die `canGoBack`-Methode bewegt, um den Tooltip zu öffnen, der die Methode beschreibt.



3 Machen Sie sich keine Gedanken, wenn Sie mit Header-Dateien nicht vertraut sind. Abschnitt 8.4, *Ein Outlet und eine Aktion deklarieren*, auf Seite 140 und das dorthin führende Material sollten Ihnen eine gute Vorstellung davon vermitteln, wie und warum sie verwendet werden.

- *Companion Guide*: Apple bietet häufig ein oder mehrere umfassende Dokumente an, die Beispiele für die Verwendung der Klasse für eine bestimmte Programmieraufgabe enthalten. In diesem Fall erhalten wir einen Verweis auf den *Web Kit Objective-C Programming Guide*.

Neben diesem Tasks-Listing aller Methoden finden Sie auch eine Liste aller Klassenmethoden sowie eine Liste aller Instanzmethoden. Es gibt auch Listen aller Konstanten und Notifikationen.

Werfen wir einen kurzen Blick auf das Listing für eine Methode.

Wenn Sie die `goBack`-Methode anklicken (nicht die `goBack:`-Methode darunter), sehen Sie Folgendes:

goBack

Loads the previous location in the back-forward list.

```
- (BOOL)goBack
```

Return Value
YES if able to move backward; otherwise, NO.

Availability
Available in Mac OS X v10.2 with Safari 1.0 and later.
Available in Mac OS X v10.2.7 and later.

See Also
- backForwardList
- goForward
- goToBackForwardItem:

Declared In
WebView.h

Sie erhalten eine kurze Beschreibung zur Funktion dieser Methode, gefolgt von der Methodensignatur:

```
-(BOOL) goBack
```

Das Minuszeichen (-) zeigt an, dass `goBack` eine Instanzmethode ist. Ein Pluszeichen (+) kennzeichnet eine Klassenmethode. Wir haben noch nicht viel über Objekte und Klassen geredet, aber wenn es so weit ist, werden Sie sehen, dass Sie Klassenmethoden für die `WebView`-Klasse nutzen, und Instanzmethoden für Objekte vom Typ `WebView`.

Der Typ des Rückgabewerts von `goBack` ist `BOOL`. Wie Sie der Beschreibung entnehmen können, lauten die beiden booleschen Werte bei Objective-C `YES` und `NO` und nicht `true` (wahr) und `false` (falsch). Abgesehen davon, dass das die Art und Weise ist, in der erfahrende Objective-C-Programmierer Code schreiben, sollten Sie ebenfalls `YES` und `NO` verwenden, damit sich Ihr Code gut lesen lässt.

3.3 Methoden mit Argumenten

Wenn Sie sich die Liste der Tasks für *Forward* und *Back* genauer ansehen, sollten Sie zwei Methoden für *Back* sehen, die nahezu identisch lauten: Es gibt die `goBack`-Methode, die wir uns gerade angesehen haben, und die `goBack:`-Methode.

Diese beiden Methoden sind völlig verschieden. Der angehängte Doppelpunkt der `goBack:`-Methode zeigt an, dass sie ein einzelnes Argument verlangt.

Schlagen Sie die `goBack:`-Methode in der `WebView`-Klassenreferenz nach. Ihre Signatur unterscheidet sich deutlich von der `goBack`-Methode ohne Argumente:

goBack:

An action method that loads the previous location in the back-forward list.

```
- (void)goBack:(id)sender
```

Parameters
sender
The object that sent this message.

Discussion
This method does nothing if it is unable to move backward.

Availability
Available in Mac OS X v10.2 with Safari 1.0 and later.
Available in Mac OS X v10.2.7 and later.

See Also
– goForward:

Declared In
WebView.h

Als Sie den *Back*-Button mit dem Web-View verknüpften, haben Sie diese `goBack:`-Methode als Aktion gewählt. Sie werden diese Art der Signatur häufig bei Methoden finden, die von GUI-Komponenten aufgerufen werden. Das `sender`-Argument ist ein Handle auf das aufrufende Objekt. Der `sender`-Typ wird per *cast* zu einer `id`. Das ist für Cocoa der generische Typ. Jeder Zeiger auf ein Objekt ist zumindest vom Typ `id`. Mit einem Handle auf den Sender können wir mit dem Objekt kommunizieren, das `goBack:` aufgerufen hat, und zwar unabhängig von seinem Typ. Ein typisches Beispiel dafür, wie man das zu seinem Vorteil nutzt, sehen wir uns in Abschnitt 3.6, *Verlinkung auf sich selbst*, auf Seite 47 an.[4]

4 Der Rückgabetyp für `goBack:` ist `IBAction`. Erst in Kapitel 8, *Einen Controller entwickeln*, auf Seite 135 werden wir darüber sprechen, was das bedeutet. Für den Moment sollten Sie ihn als `void` betrachten.

Hier der Aufruf einer Methode mit einem einzelnen Argument:

```
[myWebView goBack:self]
```

Wenn Sie immer noch zwischen Objective-C und einer anderen Sprache hin- und herübersetzen, dann ist Ihnen wahrscheinlich Folgendes vertraut:

```
myWebView.goBack(this)
```

Lassen Sie uns eine Methode mit mehreren Argumenten verwenden, damit Sie die Unterschiede besser verstehen. `WebView` besitzt eine Methode mit dem recht langen Namen `searchFor:direction:caseSensitive:wrap:`. Das ist der vollständige Methodenname. In anderen Sprachen könnte diese Methode `searchFor` heißen, aber bei Objective-C wird die Beschreibung aller Argumente als Teil des Methodennamens mit eingebunden. Zerlegen Sie die Methode in die Teile, die mit einem Doppelpunkt enden. Wenn Sie die Methode aufrufen, müssen Sie für jeden Doppelpunkt ein Argument übergeben. Die Methodensignatur verdeutlicht das, weil Sie den Rückgabewert sowie den Typ jedes Arguments angibt.

```
-(BOOL)searchFor:(NSString *)string
       direction:(BOOL)forward
   caseSensitive:(BOOL)caseFlag
            wrap:(BOOL)wrapFlag
```

Nutzen können Sie die Methode so:

```
[myWebView searchFor:myString direction:YES caseSensitive:NO wrap:YES]
```

Das Gegenstück in Java oder C# könnte etwa so aussehen:

```
myWebView.searchFor(myString, true, false, true)
```

Objective-C-Neulinge werden eher von den Doppelpunkten und der Vermischung der Argumente abgeschreckt als von den eckigen Klammern. Das verstehe ich vollkommen. Ich denke, dass ein Teil des Problems auftritt, wenn die Methode in eine einzelne Zeile passt. Vielleicht fällt Ihnen das Lesen leichter, wenn Sie eine eher vertikale Ausrichtung nutzen:

```
[myWebView searchFor:myString
           direction:YES
       caseSensitive:NO
                wrap:YES]
```

Hier folgen wird dem Objective-C-Standard zur Formatierung von Code, d. h., wir richten den Code an den Doppelpunkten aus. Das vereinfacht die Erkennung des Methodennamens als Kombination der Elemente auf der linken Seite. Gleichzeitig sind alle Argumente rechts vom Doppelpunkt zu finden.

Die Objective-C-Version erinnert vielleicht an benannte Parameter, aber das ist nicht der Fall. Die Reihenfolge der Argumente kann nicht verändert werden, und Sie können keine Argumente weglassen. Der Methodenname lautet `searchFor:direction:caseSensitive:wrap:`. Man bezeichnet ihn auch als *Selektor*, weil er zur Laufzeit genutzt wird, um die auszuführende Methode auszuwählen (also „zu selektieren").

Sie werden zu schätzen wissen, dass Sie sich nicht daran erinnern müssen, was die Parameter `true`, `false`, und `true` zu bedeuten haben (wie das bei den Java- oder C#-Versionen der Fall ist). In der Objective-C-Version wissen Sie genau, dass Sie nach einem String suchen, dass dabei die Groß- und Kleinschreibung keine Rolle spielt, dass Sie vorwärts suchen und dass das Wrapping aktiviert ist.

Ihr Cocoa-Code sollte leserlich sein. Einen Monat nachdem Sie die Methode geschrieben haben, sollten Sie immer noch in der Lage sein, Ihre Absicht zu erkennen und zu verstehen, was die Methode macht. In `WebView` finden Sie die Methode `moveToBeginningOfSentenceAndModifySelection:`. Der Name einer Methode kann länger sein als ihre Implementierung – was die Methode genau macht, ist für jeden sofort ersichtlich, der sie im Programm aufruft oder im Interface Builder eine Verbindung zu ihr herstellt.

Weitere Informationen zu Methoden finden Sie im Kapitel „Objects, Classes, and Messaging" von Apples *The Objective-C Programming Language* [App09f]. Es gibt auch ein schönes Posting zu Stack-Überläufen bei der Übergabe mehrerer Parameter in Objective-C.[5] Für Richtlinien zur Benennung von Methoden sollten Sie sich auch Apples *Coding Guidelines for Cocoa* [App06] ansehen. Matt Gallagher hat auf *Cocoa with Love* auch ein nettes Posting über Methodennamen veröffentlicht.[6]

3.4 Dynamische Bindung

Sehen wir uns an, was hinter den Kulissen passiert, wenn eine Objective-C-Nachricht gesendet wird. Der einfachste Fall ist hier zu sehen:

```
[myWebView goBack]
```

Das wird zur Laufzeit in den folgenden Funktionaufruf übersetzt:

```
objc_msgSend(myWebView, goBack)
```

5 http://stackoverow.com/questions/722651/how-do-i-pass-multiple-parameters-in-objective-c

6 http://cocoawithlove.com/2009/06/method-names-in-objective-c.html

Der Empfänger wird als erstes Argument übergeben, der Selektor als zweites. Hier die etwas kompliziertere Nachricht:

```
[myWebView searchFor:myString direction:YES caseSensitive:NO wrap:YES]
```

Das wird zur Laufzeit in diesen Funktionsaufruf umgewandelt:

```
objc_msgSend(myWebView, searchFor:direction:caseSensitive:wrap:,
                                              myString, YES, NO, YES)
```

Erneut wird der Empfänger im ersten und der Selektor im zweiten Argument angegeben. Die Parameter werden als weitere Funktionsargumente übergeben.

Es folgt eine Erklärung, die genauer ist, als Sie sie eigentlich brauchen: Zur Laufzeit wird der Selektor mit den Einträgen der Dispatch-Tabelle für die Objektklasse verglichen. Diese Tabelle zeigt auf die Speicheradresse der Prozedur, die die angeforderte Methode implementiert. Existiert kein passender Selektor in dieser Klasse, geht die Suche in der Superklasse dieser Klasse weiter und dann den Baum immer weiter nach oben bis zur Rootklasse. Weitere Informationen finden Sie in Apples *The Objective-C 2.0 Programming Language*, das in Xcode über das Dokumentationsfenster zugänglich ist.

3.5 Probleme beim Senden von Nachrichten

Es gibt zwei grundsätzliche Dinge, die bei Objective-C mit einer einfachen Nachricht schiefgehen können: Entweder existiert der Empfänger nicht, in diesem Fall hat er den Wert `nil`, oder das Objekt ist gültig, versteht aber die gesendete Nachricht nicht. Sie erhalten während der Kompilierung der Anwendung eine Warnung vom Compiler, doch keiner dieser Fehler bricht die Kompilierung ab oder unterbindet die Ausführung der Anwendung.

Im ersten Fall senden Sie eine Nachricht der Form

```
[nil irgendeineNachricht];
```

Sie erhalten keine Fehlermeldung während der Kompilierung und keine Exception zur Laufzeit. Liefert `irgendeineNachricht` darüber hinaus ein Objekt, einen Zeigertyp oder die meisten numerischen Typen zurück, dann gibt die Nachricht `0` zurück.

Im zweiten Fall senden Sie die folgende Nachricht:

```
[gültigesObjekt eineNachrichtDieKeinerVersteht];
```

Diesmal bricht die Anwendung ab, wenn dem Empfänger eine Nachricht gesendet wird, die er nicht versteht.[7] Dieses Problem ist während der Kompilierung durchgerutscht, führt zur Laufzeit aber zu einer Exception. Ist `gültigesObjekt` etwa eine Instanz der fiktiven Klasse `EigeneKlasse`, sehen Sie in der Konsole eine Meldung wie diese:

```
*** -[EigeneKlasse eineNachrichtDieKeinerVersteht]:
                    unrecognized selector sent to instance 0x109280
*** Terminating app due to uncaught exception 'NSInvalidArgumentException'
```

Das teilt Ihnen mit, dass die Nachricht `eineNachrichtDieKeinerVersteht` an eine Instanz von `EigeneKlasse` gesendet wurde, `EigeneKlasse` diese Methode aber nicht implementiert.

Weshalb wurde das also nicht während der Kompilierung abgefangen? Nun, das wurde es wohl: Sehr wahrscheinlich haben Sie eine Warnung erhalten, dass `EigeneKlasse` nicht auf die Nachricht `eineNachrichtDieKeinerVersteht` reagiert.[8]

Es gibt Zeiten, zu denen wir die Vorteile dynamischer Typisierung und dynamischer Bindung nutzen, und es gibt Zeiten, wo wir die Hilfe des Compilers zu schätzen wissen, wenn er die von uns angesprochenen Typen kennt. Ein Weg, Laufzeitfehler zu vermeiden, besteht darin, sich mit Apples Dokumentation vertraut zu machen, damit man Objekten Nachrichten sendet, die diese auch verstehen.

3.6 Verlinkung auf sich selbst

Wir haben die `WebView`-Methode `takeStringURLFrom:` benutzt, um die vom Benutzer im Textfeld eingegebene URL für den Web-View zu verwenden. Die Signatur von `takeStringURLFrom:` erinnert stark an die der `goBack:`-Methode.

```
-(IBAction)takeStringURLFrom:(id)sender
```

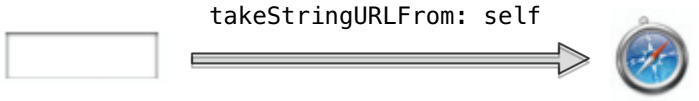
Nehmen wir an, dass es eine Instanz von `WebView` namens `myWebView` gibt, und dass das Textfeld eine Instanz von `NSTextField` namens `addressField` ist. Gibt der Benutzer einen String in das Textfeld ein und drückt ↵, wird folgende Nachricht gesendet:

```
[myWebView takeStringURLFrom: self]
```

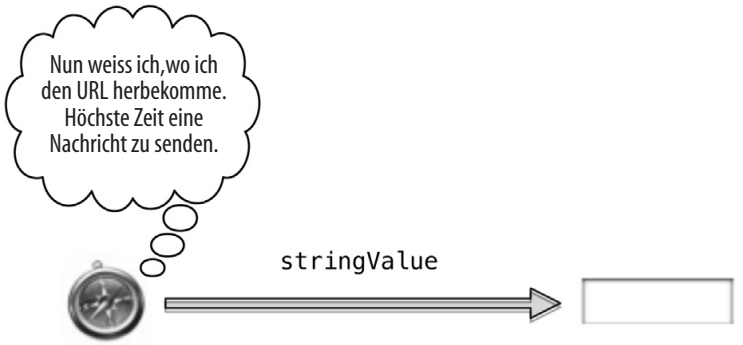
7 Ein Objekt „versteht“ eine Nachricht, wenn es (oder eine seiner Superklassen) die entsprechende Methode deklarieren und implementieren.

8 Es gibt Zeitpunkte, zu denen Sie diese Warnung ignorieren können. Sie wissen dann, dass `gültigesObjekt` keine Instanz von `EigeneKlasse` ist, sondern eine Instanz einer Klasse, die mit `eineNachrichtDieKeinerVersteht` umgehen kann.

Man kann sich das visuell als Nachricht vorstellen, die vom Textfeld an die `WebView`-Instanz gesendet wird.



Der Web-View empfängt die Nachricht und bereitet das Laden einer neuen Webseite vor. Zuerst muss er den String mit der URL irgendwoher bekommen. Aufgrund der gerade empfangenen Nachricht weiß er, woher er ihn bekommt. Das `WebView`-Objekt sendet eine Nachricht an das Objekt zurück, das die `takeStringURLFrom:`-Nachricht sendet, und fragt nach dem Stringwert.



Das `WebView`-Objekt sendet folgende Nachricht:

```
[sender stringValue]
```

Der Web-View schickt eine Nachricht an das Objekt zurück, das die `takeString-URLFrom:`-Nachricht gesendet hat, und fragt nach dem Stringwert. Der `sender` ist ein gültiges Objekt, und wir setzen voraus, dass der Typ des Objekts auf die `stringValue`-Nachricht richtig reagiert, doch der Compiler kann das für uns nicht prüfen. Wir kennen den Typ des `sender` während der Kompilierung nicht und können daher nicht wissen, ob zur Laufzeit alles funktioniert.

Der Web-View versucht dann, die URL zu laden, die als Antwort auf diese Nachricht in Form eines Strings zurückgeliefert wird.

3.7 Übung: Mehrere Verbindungen

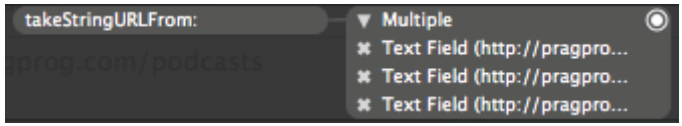
Sie können das in Aktion sehen, indem Sie das SimpleBrowser-Projekt modifizieren. Wir werden diese Version wieder verwerfen, sobald wir fertig sind. Also legen Sie eine Kopie des existierenden SimpleBrowser-Projekts an, um alle Arbeiten in dieser vorzunehmen.

Lassen Sie uns das Ganze gemeinsam durchgehen. Wenn wir fertig sind, haben Sie den Web-View mit mehr als einem Textfeld verknüpft und um die Fähigkeit erweitert, den URL-String von jedem dieser Textfelder abzurufen.

Öffnen Sie die *.nib*-Datei der Kopie Ihres Projekts und fügen Sie zwei zusätzliche Textfelder mit unterschiedlichen Standardadressen ein. Die Anwendung sollte dann etwa so aussehen:

Im Interface Builder wählen Sie den Web-View und nutzen den Inspector, um sich seine Verbindungen anzusehen. Es sollte eine Verbindung zwischen der empfangenen Aktion `takeStringURLFrom:` und dem ersten Textfeld geben, und der Kreis sollte ausgefüllt sein.

Doch wir wissen, wie `takeStringURLFrom:` funktioniert. Als Erstes sendet es eine Nachricht an die Komponente zurück, die die Methode aufgerufen hat. Es gibt also keinen Grund dafür, dass wir diese empfangene Aktion nicht mit mehreren Elementen verknüpfen könnten. Bei gedrückter Maustaste ziehen Sie den Mauszeiger vom ausgefüllten Kreis in eines der Textfelder, die Sie am unteren Rand des Fenster hinzugefügt haben. Wiederholen Sie diesen Vorgang für das zweite von Ihnen hinzugefügte Textfeld. Ihr Web-View ist jetzt in der Lage, die URL von jedem dieser Textfelder anzunehmen. Diese Mehrfachauswahl sehen Sie hier:



Speichern Sie Ihre Arbeit und testen Sie die modifizierte Anwendung. Sie können nun in jedem Textfeld URLs eingeben. Tatsächlich hätten Sie die visuellen Komponenten, mit denen Sie die URL übergeben, beliebig mischen können. Die Komponente muss nur in der Lage sein, auf die Nachricht `stringValue` mit einem String zu antworten, der eine gültige URL enthält.

Bevor wir weitermachen, können Sie die Kopie dieses Projekts speichern oder zur Ursprungsversion zurückkehren. Schließen Sie das Projekt jetzt; wir werden es in den nächsten drei Kapiteln nicht mehr benötigen.

Nachdem Sie nun wissen, wie man Methoden in Objective-C liest, können Sie eigene schreiben, um das Verhalten des SimpleBrowser anzupassen. Bevor wir zu unserem Browser zurückkehren, wollen wir uns aber etwas Zeit nehmen, um uns mit Klassen, Objekten, Instanzvariablen und Eigenschaften (Properties) zu beschäftigen.

Kapitel 4

Klassen und Objekte

Für die nächsten Kapitel legen wir unser Webbrowser-Beispiel beiseite. Nachdem Sie nun Methoden lesen können, wird es Zeit, selbst einige zu schreiben. In diesem Kapitel beginnen wir damit, Code in Objective-C zu schreiben. Wir fangen mit einer grundlegenden „Hallo, Welt!"-Anwendung an und lernen, wie man Klassen, Objekte, Variablen und Methoden erzeugt.

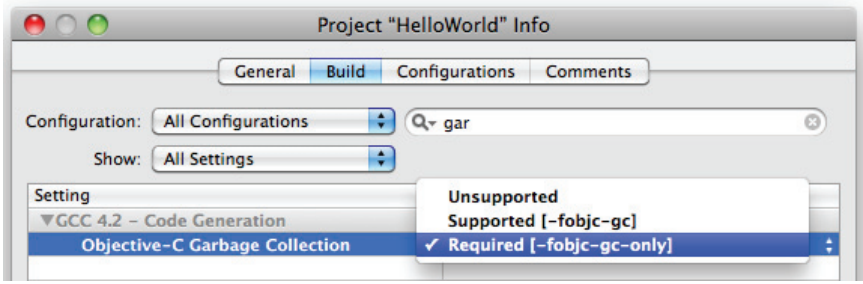
Denken Sie daran, dass es hier nicht darum geht, eine tolle „Hallo, Welt!"-Anwendung zu entwickeln. Die Ausgabe des Programms wird sich im Verlauf des Kapitels nicht wesentlich verändern. Hier geht es um Fortschritt durch die Nutzung verschiedener Techniken. Sie werden eine von Apples Klassen benutzen und dann eine eigene aufbauen. Sie werden mit diesen Klassen über Klassenmethoden kommunizieren. Sie werden dann eine Instanz einer Klasse erzeugen und Instanzmethoden benutzen, um mit diesen Objekten zu kommunizieren. Nachdem wir unsere Selbstachtung als *echte* Entwickler wiedergewonnen haben, kehren wir in Kapitel 8, *Einen Controller entwickeln*, auf Seite 135 zu unserem Webbrowser zurück und passen dessen Verhalten über selbst entwickelten Code an.

4.1 „Hallo, Welt!"

Lassen Sie uns unser „Hallo, Welt!"-Projekt in Xcode anlegen. Sie wissen ja noch, dass neue Projekte in Xcode mit `File > New Project` oder ⇧⌘N angelegt werden. Wählen Sie die Vorlage `Application > Cocoa Application`, lassen Sie die Checkboxen deaktiviert und nennen Sie das Projekt *HelloWorld*. Klicken Sie *Build & Run* an, und nach einer kurzen Pause erscheint ein leeres Fenster. Die Anwendung funktio-

niert, sie macht nur nichts. Beenden Sie HelloWorld oder halten Sie die Tasks aus Xcode heraus an, indem Sie das Stopzeichen anklicken, `Run > Stop` wählen oder ⇧⌘↵ drücken.

Wir werden vor Kapitel 6, *Speicher*, auf Seite 95 nicht weiter auf Speicher eingehen. Dennoch möchte ich, dass Sie sich angewöhnen, die automatische Garbage Collection für jedes Cocoa-Projekt zu aktivieren. In Xcode wählen Sie `Project > Edit Project Settings`. Das öffnet das Fenster mit den Projekteinstellungen (Project Settings). Wählen Sie den Build-Reiter und geben Sie das Wort *garbage* in den Filter ein. Sie müssen nicht das ganze Wort eingeben, damit die Einstellung *Objective-C Garbage Collection* zu sehen ist.



Wählen Sie in der Drop-down-Liste *Required* aus. Schließen Sie die Einstellungen, und schon unterstützt Ihr Projekt die automatische Garbage Collection.

4.2 Logging von Ausgaben an die Konsole

Im Moment wollen wir die GUI ignorieren und über `NSLog()` direkt auf die Konsole schreiben. Wir übergeben einen `NSString` an `NSLog()`, der im Konsolenfenster ausgegeben werden soll.[1]

Statt den auszugebenden String einfach nur in Anführungszeichen zu stellen, müssen Sie einen `NSString` wie folgt angeben:

```
NSLog(@"Hello, World!");
```

Mit anderen Worten müssen Sie ein `@` vor dem öffnenden Anführungszeichen angeben, um den Anfang eines Cocoa-`NSString` anzuzeigen.

1 Viele Core-Klassen wie `NSString` beginnen mit *NS*, was für „Apple“ steht. Eigentlich steht NS für „NeXTSTEP“, und diese Klassen wurden nicht umbenannt, nachdem Apple NeXT gekauft hatte. Objective-C kennt keine Namensräume, weshalb Klassen häufig mit einem aus zwei oder drei Buchstaben bestehenden Identifier beginnen. Der gleichen Konvention folgen auch viele C-Funktionen wie `NSLog()`.

Wenn Sie das @ vergessen, erhalten Sie vom Compiler eine Warnung, dass Sie ein Argument mit einem inkompatiblen Zeigertyp übergeben. Leider ist die Syntax-Einfärbung für Anführungszeichen mit und ohne @ identisch, weshalb es nicht direkt ins Auge springt, wenn das @ fehlt.

Bei traditionellen in C geschriebenen „Hallo, Welt!"-Versionen (aber auch bei älteren Cocoa-Varianten) würde diese Zeile in `main()` stehen. Ihr Programm besitzt ein `main()` in der Datei `main.m`, die Sie unter *Groups & Files* im Verzeichnis `Other Sources` finden.

Funktionen

Bei `NSLog( )` sieht nichts so aus wie die eckigen Klammern, Doppelpunkte, Ziele und Aktionen, die wir im vorigen Kapitel gesehen haben.

Das soll auch nicht so sein. `NSLog()` ist keine Objective-C-Methode, sondern eine C-Funktion. Objective-C setzt auf C auf. Sie sollten zwar den Objective-C-Stil des Sendens von Nachrichten an Klassen und Objekte bevorzugen, aber Sie können das gute alte C benutzen, um Dinge zu erledigen.

Sie werden viele Funktionen im C-Stil sehen, wenn Sie Aufrufe auf Systemebene nutzen und mit APIs arbeiten, die mit grafischen Objekten umgehen. Sie erkennen die C-Funktionen daran, dass sie ihre Argumente in gewöhnlichen runden Klammern einschließen und die Funktion ohne Ziel aufgerufen wird. In diesem Kapitel werde ich hier und da auf die Unterschiede eingehen, und schon bald werden Sie sich an die Mischung von C- und Objective-C-Konstrukten gewöhnt haben.

Classes/HelloWorld1/main.m

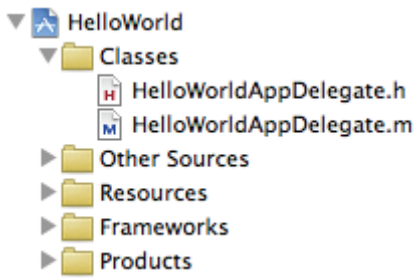
```
#import <Cocoa/Cocoa.h>
int main(int argc, char *argv[])
{
     return NSApplicationMain(argc,(const char **) argv);
}
```

Wir werden aber unsere Finger von `main.m` lassen. Wir werden sie überhaupt nie modifizieren oder etwas hinzufügen. Alle Aktionen für `main( )` wurden in den Aufruf der Funktion `NSApplicationMain( )` gepackt. In unserem Fall erzeugt diese Funktion eine Instanz der Klasse `NSApplication` und lädt das `MainMenu`-nib. Sie startet mit anderen Worten die Anwendung.

Als C-Programmierer werden Sie versucht sein, einen größeren Teil der Programmierlogik in der `main()`-Funktion unterzubringen. Machen Sie das nicht! Diese Anpassungen finden nun an anderer Stelle statt, dem Application Delegate (also dem „Anwendungsdelegierten“). Sie werden im Verlauf dieses Buches viele dieser Delegate-Muster sehen.

Bei diesem ersten Durchgang werde ich nicht auf die Details eingehen, aber das Grundprinzip ist, dass die `NSApplication`-Klasse weiß, wie sich eine Anwendung im Allgemeinen verhalten sollte. Eine Lösung könnte für uns darin bestehen, dass wir eine Subklasse von `NSApplication` anlegen und die Methoden überschreiben, deren Verhalten wir anpassen möchten. Bei Cocoa-Anwendungen neigt man aber zu einem anderen Ansatz: Wir weisen `NSApplication` ein Delegate-Objekt zu und implementieren nur die Methoden, die wir benötigen. Und was ist ein Delegate? Sie können sich ihn als eine Art Assistentenklasse vorstellen, an die die Hauptklasse die Arbeit übergibt. Wenn die Hauptklasse, in diesem Fall `Application`, etwas erledigen möchte, ruft sie eine Methode ihrer Delegate-Klasse auf.

Glücklicherweise hat die Cocoa Application-Vorlage für uns bereits einen Application Delegate namens `HelloWorldAppDelegate` angelegt. Sie finden die Quelldateien im Ordner *Classes* unter *Groups & Files*.[2]



Die Cocoa Application-Vorlage hat die Dateien `HelloWorldAppDelegate.h` und `HelloWorldAppDelegate.m` angelegt, die zusammen die Klasse `HelloWorldAppDelegate` definieren.

In `HelloWorldAppDelegate.m` finden Sie eine Methode namens `applicationDidFinishLaunching:`, die den folgenden Kommentar enthält:

```
// Insert code here to initialize your application
```

Fügen Sie die Logzeile anstelle des Kommentars ein:

2 Wenn Sie diese Dateien nicht finden können, verwenden Sie wahrscheinlich die Xcode-Version 3.1 (oder eine noch ältere Version). Sie müssen dann zu Xcode 3.2 (oder höher) wechseln.

Classes/HelloWorld1/HelloWorldAppDelegate.m

```
#import "HelloWorldAppDelegate.h"
@implementation HelloWorldAppDelegate
@synthesize window;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
   NSLog(@"Hello, World!");
}
@end
```

Und wie wird diese Methode aufgerufen? Wir rufen sie nicht explizit von `main()` aus auf. Tatsächlich rufen wir sie gar nicht explizit auf.

Die einfache Erklärung lautet, dass nach dem Start der Anwendung zur Laufzeit das Application-Objekt die Nachricht `applicationDidFinishLaunching:` an seinen Delegate sendet. Die etwas längere Version haben Sie im vorigen Kapitel gesehen. Implementiert der Delegate die Methode nicht, kommt es beim Aufruf zu einem Laufzeitfehler. Tatsächlich prüft das Application-Objekt also, ob der Delegate die Methode implementiert, und wenn das der Fall ist, sendet er eine Nachricht.

Um die Anwendung auszuführen, öffnen Sie zuerst mit `Run` > `Console` oder ⇧⌘R eine Konsole. Als Nächstes löschen Sie die Konsole über `Run` > `Clear Console`, über ⌃⌥⌘R oder durch einen Klick auf den *Clear Console*-Button (wenn er sichtbar ist). Abschließend klicken Sie auf *Build & Run* (oder nutzen das entsprechende Tastaturkürzel ⌘↵).

HelloWorld sollte nun starten: Ein leeres Fenster erscheint und die Konsole enthält eine Zeile mit der Uhrzeit, zu der die Session gestartet wurde, gefolgt von einer Zeile mit dem Timestamp, der Prozess-ID und unserem String. Meine Ausgabe sieht so aus:

```
HelloWorld[19673:10b] Hello, world!
```

Sobald Sie Ihre Leistung genug bewundert haben, können Sie HelloWorld beenden und wieder an die Arbeit gehen.

4.3 Eine vorhandene Klasse nutzen

Wir haben eine „Hello, World“-App entwickelt, die etwas auf der Konsole ausgibt. Im nächsten Schritt wollen wir mit Programmcode ein Textfeld erzeugen, in unser Fenster einfügen, ein wenig konfigurieren und „Hello, World!“ in diesem Textfeld ausgeben. Denken Sie daran, dass unser eigentliches Ziel nicht darin besteht, „Hello, World!“ in einem Textfeld auszugeben. Wenn das unser Ziel gewesen wäre, hätten wir einfach im Interface Builder das Widget aus der Library nehmen und *Hello, World!* eintragen können.[3] Die Faustregel für den Interface Builder ist einfach: Wann immer wir ihn nutzen können, *sollten* wir den Interface Builder auch nutzen. Alles, was Sie mit dem Interface Builder machen, können Sie auch mit Programmcode erreichen. Außer in ein paar sehr geheimnisvollen Fällen gibt es aber keinen Grund dazu, den Interface Builder zu meiden.

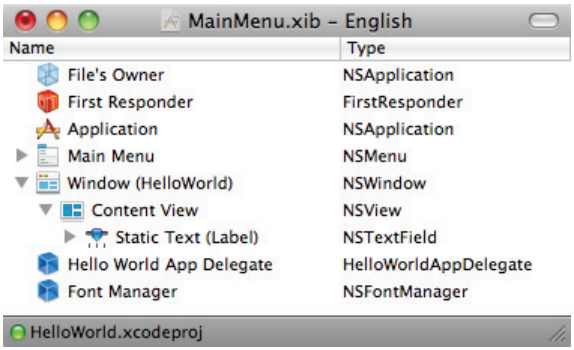
Das hier ist einer dieser Fälle. Unser Ziel besteht in diesem Abschnitt darin, Erfahrung damit zu sammeln, wie man eine Klasse, die man nicht selbst geschrieben hat, per Programmcode instanziiert, konfiguriert und nutzt. Danach sind wir so weit, unsere eigene Klasse zu entwickeln und zu verwenden.

Um die Sache etwas zu vereinfachen, klicken wir `MainMenu.xib` doppelt an, um den Interface Builder zu starten, wählen unser Fenster aus und nutzen den Size Inspector, um die Breite des Fensters auf 580 und die Höhe auf 90 Pixel festzulegen. Leider kann man den View leicht versehentlich auswählen, wenn man zufällig auf das Fenster klickt. Wenn Sie die Titelleiste für das Fenster anklicken, sollte der Size Inspector *Window Size* anzeigen.

Ich habe die minimale und maximale Größe des Fensters auf diese Werte gesetzt und gleichzeitig festgelegt, dass der Benutzer die Größe des Fensters nicht verändern kann. Diese Größe hat keine besondere Bedeutung; aber es hilft uns beim Erzeugen und Plazieren des Labels, wenn wir die Größe des Fensters kennen.

Erinnern Sie sich nun daran, was passiert ist, als wir in unserem SimpleBrowser-Beispiel das Textfeld über dem Fenster platzierten. Wir haben das Textfeld aus der Library in das Fenster gezogen und dann die Maustaste losgelassen. Es wurde automatisch in die View-Hierarchie eingefügt. Ihr Dokumentenfenster sah etwa so aus:

3 Eine positive Nebenwirkung davon, dass wir in diesem Beispiel ein Textfeld mit Programmcode erzeugen, besteht darin, dass es Sie davon überzeugen wird, wann immer möglich den Interface Builder zu nutzen.



Auch wenn Sie damals wahrscheinlich nicht darauf geachtet haben, enthielt das Fenster einen `ContentView`. Als Sie das Textfeld in das Fenster zogen, wurde eine Instanz von `NSTextField` erzeugt und als Subview in den `ContentView` eingefügt.

Sie haben das Textfeld dann positioniert und den gewünschten Text eingegeben. Mit Programmcode könnte das wie folgt aussehen:[4]

Classes/HelloWorld2/HelloWorldAppDelegate.m

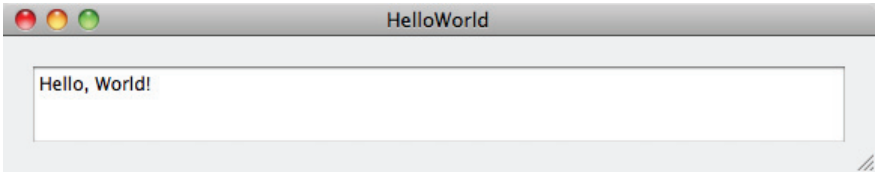
```
#import "HelloWorldAppDelegate.h"
@implementation HelloWorldAppDelegate
@synthesize window;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSRect labelLocation = NSMakeRect(20, 20, 540, 50);
    NSTextField *label = [[NSTextField alloc] initWithFrame:labelLocation];
    [label setStringValue:@"Hello, World!"];
    [[self.window contentView] addSubview:label];
}
@end
```

Zuerst erzeugen Sie ein `NSRect`, das die Information über Größe und Lage des Labels enthält. Das entspricht unserer Arbeit mit dem Interface Builder in Kapitel 2, *Vorhandenes nutzen*, auf Seite 11, wo wir die Größe und Lage des Textfelds mit dem Size Inspector angepasst haben. Beachten Sie, dass `NSMakeRect( )` eine C-Funktion ist, keine Methode. Das ist durchaus üblich, wenn Sie mit grafischen Elementen arbeiten. Auch wenn das im Augenblick eigentlich zu viele Informationen sind: `NSRect` und `NSPoint` sind C-Strukturen, keine Objective-C-Objekte. Es gibt immer eine entsprechende C-Funktion wie `NSMakeRect()` oder `NSMakePoint()`, um diese Elemente zu erzeugen und zu konfigurieren.

4 Ich setze voraus, dass Sie wissen, dass `self.window` auf die `window`-Instanz des aktuellen Objekts verweist. Ich erkläre die Punktsyntax in Kapitel 5, *Instanzvariablen und Eigenschaften* auf Seite 77.

Als Nächstes erzeugen wir eine `NSTextField`-Instanz namens `label` und setzen ihre Lage und Größe auf die Werte, die wir in der Variablen `labelLocation` festgelegt haben. Dann legen wir den Stringwert des Labels mit `Hello, World!` fest und fügen das Label als Subview in den Content-View des Fensters ein.

Klicken Sie auf *Build & Run*. Sie sehen Folgendes:



Ich bin mit dem aktuellen Zustand unseres Projekts nicht besonders glücklich. Die Ausgabe sieht zwar nicht besonders gut aus, was wir noch korrigieren werden, aber hauptsächlich geht es mir um den Zustand des Codes. Lassen Sie uns ein wenig refaktorieren.

4.4 Code refaktorieren

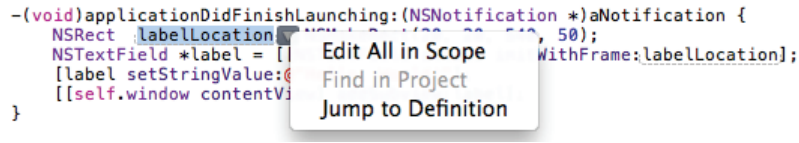
Hin und wieder ist es hilfreich, wenn man einen Schritt zurücktritt und schaut, wo man noch ein wenig aufräumen könnte. Wenn ich ihn mir so ansehe, gefällt mir der Name `labelLocation` eigentlich nicht. Natürlich könnten wir die Variable völlig weglassen und die Erzeugung von `NSRect` direkt einbinden, ohne eine (selbst-)erklärende temporäre Variable zu nutzen. Das wäre eine Lösung.

Wenn wir diese Variable umbenennen, müssen wir bedenken, dass es nicht nur um die Lage des Labels geht, sondern auch um seine Größe. Wie Sie aus dem Methodennamen `initWithFrame` ableiten können, werden diese beiden Informationen in einem `NSRect` gesammelt und und als *Frame* bezeichnet.

Lassen Sie uns Xcode nutzen, um den Variablennamen von `labelLocation` in `label-Frame` zu ändern. In diesem Fall hätten wir das natürlich einfach von Hand machen können, aber ich möchte Ihnen zeigen, wie Sie Xcode benutzen können, um eine Reihe von Refaktorierungen vorzunehmen.

Ich weiß nicht, ob Sie es bemerkt haben, aber wenn Sie `labelLocation` anklicken (Na los, trauen Sie sich!), werden alle Instanzen von `labelLocation` unterstrichen.[5] Wenn Sie die Maus leicht rechts neben die von Ihnen gewählte `labelLocation` bewegen, erscheint ein Drop-down-

Pfeil. Das ist genau die Schnittstelle, die Sie sehen, wenn Sie die Maus in `Mail.app` über eine Telefonnummer oder eine Adresse bewegen. Klicken Sie den Pfeil an, und Sie sollten die folgenden Optionen sehen:



Wählen Sie *Edit All in Scope* („alles im Bereich editieren“). Beide Instanzen von `labelLocation` sollten hervorgehoben werden. Ändern Sie an einer Stelle *Location* in *Frame*, und Sie werden sehen, dass beide in `labelFrame` geändert werden.

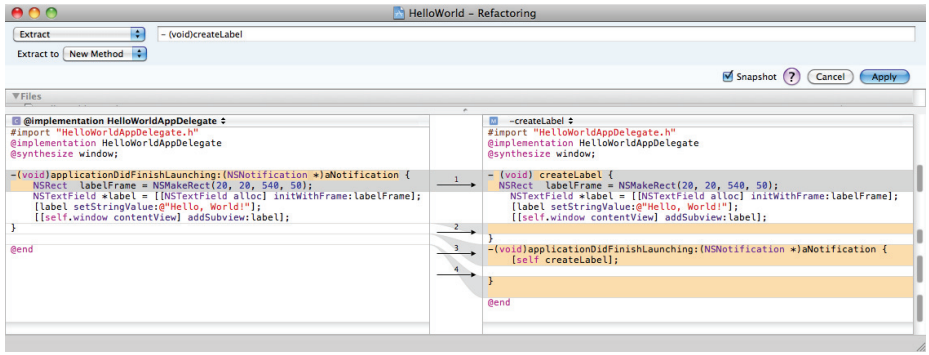
Als Nächstes kann (auch wenn es im Moment nicht nach einem Problem aussieht) die Methode `applicationDidFinishLaunching:` aus dem Ruder laufen. Ich möchte mit kleinen, zusammenhängenden, einfach zu verstehenden Methoden arbeiten. Lassen Sie uns eine schnelle Refaktorierung vornehmen und die folgenden vier Zeilen in eine separate Methode packen:

```
NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
NSTextField *label = [[NSTextField alloc] initWithFrame:labelFrame];
[label setStringValue:@"Hello, World!"];
[[self.window contentView] addSubview:label];
```

Auch diese Refaktorierung könnten wir leicht von Hand vornehmen, aber wir wollen die Refactoring-Tools von Xcode verwenden. Markieren Sie genau diese vier Zeilen und wählen Sie dann `Edit > Refactor ...`, drücken Sie ⇧⌘J oder Control-klicken Sie und wählen dann Refactor... aus dem erscheinenden Menü. Die Refactoring-Dialogbox erscheint und bieten Ihnen nach der Analyse des markierten Codes eine Drop-down-Liste mit verfügbaren Refaktorierungen.

Wählen Sie *Extract*. Die voreinstellte Signatur der neuen Methode ist `-(void) extracted_method`. Wir ändern das in `-(void) createLabel`. Klicken Sie den *Preview*-Button an. Sie erhalten eine Liste aller Dateien, in denen es Änderungen gibt. In diesem Fall ergeben sich durch diese Refaktorierung nur Änderungen in `HelloWorldAppDelegate.m`. Vier Änderungen werden angegeben. Klicken Sie `HelloWorldAppDelegate.m` im Refactoring-Fenster an, und Sie sollten diese Vorschau sehen:

5 Hier erscheint `labelLocation` nur einmal, aber Sie verstehen, worum es geht.



Klicken Sie den *Apply*-Button an, um die Refaktorierung abzuschließen.

Hier die sich daraus ergebende `HelloWorldAppDelegate.m`:

Classes/HelloWorld3/HelloWorldAppDelegate.m

```
#import "HelloWorldAppDelegate.h"
@implementation HelloWorldAppDelegate
@synthesize window;

-(void)createLabel {
   NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
   NSTextField *label = [[NSTextField alloc] initWithFrame:labelFrame];
   [label setStringValue:@"Hello, World!"];
   [[self.window contentView] addSubview:label];
}
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    [self createLabel]; }
@end
```

Jetzt sieht `applicationDidFinishLaunching:` schön und sauber aus. Sie besteht nur aus einem Aufruf von `createLabel`. Hier noch vier kurze Anmerkungen:

- Es gibt keinen Grund dazu, `createLabel` als Methode auszulegen. Ich hätte mich während der Refaktorierung auch dafür entscheiden können, sie als gewöhnliche C-Funktion zu implementieren.

- Denken Sie daran, dass wir Methoden aufrufen, indem wir Nachrichten an Objekte senden. In unserem Fall ist `self` das Ziel der Nachricht `createLabel`, das auf die aktuelle Instanz von `HelloWorldAppDelegate` verweist.

- Wäre `createLabel` stattdessen als C-Funktion definiert worden, hätten wir `[self createLabel];` durch `createLabel();` ersetzt.

- Das Refactoring-Tool platziert die neue Methode im Quellcode über der Zeile, in der die Methode aufgerufen wird. Stellen Sie sich den Compiler vor, wie er den Quellcode von oben nach unten durchgeht: Wenn er zu der Zeile gelangt, in der die Methode aufgerufen wird, muss er die Implementierung der Methode bereits kennen. Steht die Methode im Quellcode *hinter* dem Aufruf, erhalten Sie eine Warnung vom Compiler.

Nachdem ich den Code ausgelagert habe, der das Label erzeugt, füge ich noch etwas Code hinzu, um es so zu konfigurieren, wie ich möchte. Die Anpassung durch meinen Code entspricht dabei einer Änderung, die ich im Interface Builder mithilfe des Attributes Inspector vorgenommen hätte.

Classes/HelloWorld4/HelloWorldAppDelegate.m

```
-(void) createLabel {
    NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
    NSTextField *label = [[NSTextField alloc] initWithFrame:labelFrame];
►   [label setEditable:NO];
►   [label setSelectable:NO];
►   [label setAlignment:NSCenterTextAlignment];
►   [label setFont:[NSFont boldSystemFontOfSize:36]];
    [label setStringValue:@"Hello, World!"];
    [[self.window contentView] addSubview:label];
}
```

Ich lege das Label so fest, dass es vom Benutzer nicht ausgewählt oder verändert werden kann. Ich zentriere den Text und gebe ihn fett und in 36-Punkt-Schrift aus. Die Ausgabe sieht etwas besser aus, aber einen Designpreis werden wir dafür von Apple wohl nicht bekommen:

Code wie diesen werden Sie nicht allzu oft schreiben wollen. Alles, was mit dem Interface Builder möglich ist, lässt sich auch in Code realisieren. Generell gilt aber, dass man, wenn man etwas mit dem Interface Builder erledigen *kann*, es auch mit ihm erledigen *sollte*. Ziehen Sie den Interface Builder immer dem Schreiben von Code vor.

4.5 Eine neue Klasse erzeugen

Lassen Sie uns eine eigene Klasse entwickeln. Wir werden eine Klasse namens `Greeter` entwickeln und ihr Nachrichten schicken. Dann werden wir die Klasse nutzen, um ein Objekt zu instanziieren und um diesem Objekt Nachrichten zu senden.

Alle Klassen werden in Xcode erzeugt. Instanzen der Klassen können auch im Interface Builder erzeugt werden, aber die Klassen selbst werden in Xcode erzeugt.

In Xcode wählen Sie den Ordner `Classes` unter *Groups & Files*, damit die von Ihnen angelegten Dateien in diesem Ordner erscheinen. Wählen Sie `File > New File...` oder drücken Sie ⌘N. Wählen Sie `Cocoa > Objective-C class`: diese Klasse ist eine Subklasse von `NSObject`. Die Beschreibung sagt Ihnen, das Sie „eine Objective-C-Klassendatei anlegen, mit einem optionalen Header, der den <Cocoa/Cocoa.h>-Header enthält.“ Klicken Sie auf *Next*.

Dateien organisieren

Auf der linken Seite ihres Editorfensters sollten Sie den Bereich *Groups & Files* sehen. Sie müssen eventuell das Dreieck neben dem Anwendungsnamen anklicken, um die Ordner *Classes*, *Other Sources*, *Resources*, *Frameworks* und *Products* angezeigt zu bekommen.

Wenn Sie eine neue Klasse in Xcode anlegen, erscheint die Implementierungs- und Header-Datei im gerade gewählten Ordner. Ist kein Ordner gewählt, erscheinen die Dateien auf der obersten Ebene unter dem Namen der Anwendung. Es gibt keinen Zusammenhang zwischen der Lage des Dateien auf der Festplatte und ihrem Erscheinen im Organizer. Der Organizer funktioniert eher wie Playlisten von iTunes als wie Mailboxen bei Mail.

Legen Sie bei Bedarf zusätzliche Ordner an, doch bei kleinen Projekten sollten Sie den Quellcode im *Classes*-Ordner vorhalten. Wählen Sie entweder *Classes*, bevor Sie eine neue Klasse anlegen, oder verschieben Sie die Dateien im Nachhinein in diesen Ordner.

Nennen Sie die Datei `Greeter.m`.[6] Stellen Sie sicher, dass die Checkboxen für die Erzeugung von `Greeter.h` und das Ziel `HelloWorld` aktiv sind. Im Allgemeinen sollten Sie einfach die Voreinstellungen akzeptieren können. Klicken Sie auf *Finish*. Sie haben nun die Header-Datei `Greeter.h` und die Implementierungsdatei `Greeter.m` erzeugt.

Die Header-Datei enthält das öffentliche (public) Interface für die Klasse `Greeter`. In ihr teilen Sie anderen Leuten mit, wie sie mit ihrer Klasse interagieren können. Am Anfang dieser Datei importieren Sie häufig Header-Dateien oder andere Klassen, die Ihre Klasse verwendet. In unserem Fall hat Xcode bereits die Direktive für den Import von `Cocoa.h` eingefügt, die die Header-Dateien für alle Cocoa-Klassen einbindet, die Sie möglicherweise nutzen wollen.

Hier ist die Header-Datei mit zwei Kommentaren, die ich eingefügt habe, um unsere Diskussion zu vereinfachen:

Classes/HelloWorld5/Greeter.h

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

@interface Greeter : NSObject {
     // Hier deklarieren Sie die Instanzvariablen
}
    // Hier deklarieren Sie die Methoden
@end
```

Alles zwischen `@interface` und `@end` dient der Beschreibung der öffentlichen Schnittstelle der `Greeter`-Klasse. `Greeter : NSObject` gibt an, das die Klasse `Greeter` direkt die Rootklasse `NSObject` erweitert. Im Gegensatz dazu ist `WebView` eine Subklasse von `NSView`, die wiederum eine Subklasse von `NSResponder` darstellt, die wiederum eine Subklasse von `NSObject` ist. Solange Sie sie nicht explizit überschreiben, profitieren Sie von der Vererbung des Verhaltens aller Superklassen. Das allen Objekten gemeinsame Verhalten ist in der Rootklasse `NSObject` definiert. Sie werden sich manchmal die Dokumentation einer Superklasse ansehen müssen, um die Methoden aufzuspüren, die den von Ihrer Klasse erzeugten Objekten zur Verfügung stehen.

In den Header-Datei zeigen die von mir eingefügten Kommentare, dass die Deklaration ihrer Instanzvariablen innerhalb der geschweiften

6 Beginnen Sie Klassennamen mit einem Großbuchstaben und verwenden Sie CamelCase. Beginnen Sie Variablennamen und Methoden mit einem Kleinbuchstaben. Sehen Sie sich auf jeden Fall die *Coding Guidelines for Cocoa* [App06] an, die Apples Dokumentation zur Namensgebung von Klassen, Variablen, Methoden und Funktionen enthalten.

Klammern erfolgt und die Deklaration Ihrer Methoden zwischen der schließenden geschweiften Klammer und dem `@end`.

Hier sehen Sie `Greeter.m`, die gerade generierte Implementierungsdatei:[7]

Classes/HelloWorld5/Greeter.m

```
#import "Greeter.h"

@implementation Greeter

@end
```

Die Datei beginnt mit dem Import der Header-Datei für `Greeter`. Darüber hinaus sieht man nichts anderes als die Anfangs- und Endmarkierungen für die Klassenimplementierung. Beachten Sie, dass Sie in dieser Datei nicht angeben, dass `Greeter` von `NSObject` erbt.

Sie werden diese Kombination aus `Greeter.h`- und `Greeter.m`-Dateien benutzen, um die Klasse `Greeter` zu definieren. Die Header-Datei einer Klasse enthält die Informationen, die Sie öffentlich zugänglich machen wollen. Andere Klassen werden diese Header-Datei importieren, damit sie wissen, welche Nachrichten an die Klasse gesendet werden können. Die Implementierungsdatei enthält den Teil der Klasse, der niemanden etwas angeht. Die Header-Datei teilt den anderen mit, was mit dieser Klasse bzw. mit den Objekten dieser Klasse getan werden kann. Die Implementierung versteckt die Details.

4.6 Eine Klassenmethode erzeugen und nutzen

Bei dieser Implementierung von „Hello, World!“ wollen wir eine neue Klasse definieren, aber keine Objekte dieser Klasse erzeugen. In der Praxis erzeugen wir üblicherweise ein oder mehr Objekte einer Klasse und arbeiten mit diesen Instanzen. Ich möchte, dass Sie diese Weiterentwicklung sehen, damit Sie wisssen, wie man sowohl Klassen als auch Instanzmethoden aufbaut.

Die `Greeter`-Klasse erhält die Klassenmethode `greeting`. Wir können diese Methode aufrufen, ohne eine Instanz von `Greeter` zu erzeugen. Mit anderen Worten ist das Ziel der `greeting`-Methode die Klasse `Greeter` und nicht ein Objekt des Typs `Greeter`. Der Aufruf passiert folgendermaßen:

7 Beachten Sie die Endungen der beiden Dateien `Greeter.h` und `Greeter.m`. Die `h`-Datei enthält den Header und die `m`-Datei die Implementierung. Diese und weitere interessante Fakten finden Sie in den englischsprachigen Objective-C-FAQ auf *http://www.faqs.org/faqs/computer-lang/Objective-C/faq/*.

```
[Greeter greeting];
```

Deklarieren Sie die Klassenmethode in `Greeter.h`. Vor die Methodendeklaration stellen Sie ein `+ (NSString *)`, um anzugeben, dass `greeting` eine Klassenmethode ist, die einen `NSString` zurückliefert. Sie deklarieren Ihre Klassenmethoden zwischen der schließenden geschweiften Klammer und dem `@end`.

Classes/HelloWorld6/Greeter.h

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

@interface Greeter : NSObject {
}
► +(NSString *) greeting;
@end
```

Die Methode gibt einen `NSString` zurück, der unsere Begrüßung enthält. Hier sehen Sie die Implementierungsdatei:

Classes/HelloWorld6/Greeter.m

```
#import "Greeter.h"

@implementation Greeter
► +(NSString *) greeting {
►     return @"Hello, World!";
► }
@end
```

Sie müssen zwei Änderungen an `HelloWorldAppDelegate.m` vornehmen. Zum einen muss die Header-Datei für `Greeter` eingefügt werden, damit sich der Compiler nicht beschwert, wenn Sie versuchen, `greeting` aufzurufen. Zum anderen muss der Stringwert des Labels auf den Wert gesetzt werden, der zurückgegeben werden soll, wenn Sie die Nachricht `greeting` an die Klasse `Greeter` senden.

Classes/HelloWorld6/HelloWorldAppDelegate.m

```
#import "HelloWorldAppDelegate.h"
► #import "Greeter.h"
@implementation HelloWorldAppDelegate
@synthesize window;

-(void) createLabel {
    NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
    NSTextField *label = [[NSTextField alloc] initWithFrame:labelFrame];
     [label setEditable:NO];
     [label setSelectable:NO];
     [label setAlignment:NSCenterTextAlignment];
     [label setFont:[NSFont boldSystemFontOfSize:36]];
►    [label setStringValue:[Greeter greeting]];
     [[self.window contentView] addSubview:label];
```

```
}

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    [self createLabel];
}
@end
```

Klicken Sie auf *Build & Run*, und das Ergebnis sollte genau aussehen wie das vorhin. Beenden Sie dann die Anwendung; jetzt wollen wir sie so ändern, dass sie Objekte erzeugt und Instanzmethoden nutzt.

4.7 Ein neues Objekt erzeugen

Als Nächstes wollen wir ein neues Objekt erzeugen und eine Instanzmethode aufrufen. Weil wir das Projekt mit Garbage Collection konfiguriert haben, gibt es eine ganze Reihe von Details zum Speichermanagement, um die Sie sich im Moment nicht kümmern müssen. Ich werde sie Ihnen erst in Kapitel 6, *Speicher*, ab Seite 95 vorstellen, weil die Details ein wenig trocken, aber doch wichtig sind, falls Sie an einer Cocoa-Anwendung ohne Garbage Collection oder für das iPhone arbeiten, bei der die Garbage Collection nicht unterstützt wird.[8] Weil die Garbage Collection aktiviert ist, müssen wir nur einige kleinere Änderungen an unserem Code vornehmen, um ein Objekt vom Typ `Greeter` zu erzeugen.

Zuerst machen Sie in der Header-Datei `Greeter.h` aus dem `+` ein `-`, um aus der `greeting`-Methode eine Instanzmethode anstelle einer Klassenmethode zu machen:

Classes/HelloWorld7/Greeter.h

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

@interface Greeter : NSObject {
}
► -(NSString *) greeting;
@end
```

Ändern Sie in der Datei `Greeter.m` das `+` in ein `-`, da sich der Compiler sonst darüber beschwert, dass Sie eine Instanzmethode deklariert haben, ohne sie zu implementieren. (Andererseits beschwert sich der Compiler nicht, wenn Sie eine Klassenmethode implementieren, ohne sie zu deklarieren.)

8 Tatsächlich ist das Speichermanagement nicht das erste Detail, das ich überspringe. Sie werden ein oder zwei unkommentierte Sternchen (*) gesehen haben. Ich habe Ihnen auch nicht erzählt, was passiert, wenn die Anwendung gestartet wird. Wie kann es sein, dass unser Browser direkt zu funktionieren scheint, und woher weiß das System, dass in `HelloWorldAppDelegate` unsere Objekte liegen?

Classes/HelloWorld7/Greeter.m

```
#import "Greeter.h"

@implementation Greeter
-(NSString *) greeting {
     return @"Hello, World!";
}
@end
```

Nun ist es endlich an der Zeit, unser erstes Objekt zu erzeugen und unsere erste Instanzmethode aufzurufen. Ändern Sie die `createLabel`-Methode folgendermaßen in `HelloAppDelegate.m`:

Classes/HelloWorld7/HelloWorldAppDelegate.m

```
1  -(void) createLabel {
2     NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
3     NSTextField *label = [[NSTextField alloc] initWithFrame:labelFrame];
4     [label setEditable:NO];
5     [label setSelectable:NO];
6     [label setAlignment:NSCenterTextAlignment];
7     [label setFont:[NSFont boldSystemFontOfSize:36]];
►     Greeter *greeter = [[Greeter alloc] init];
►     [label setStringValue:[greeter greeting]];
10    [[self.window contentView] addSubview:label];
11 }
```

Die rechte Seite des Codes in Zeile 8 ist der Template-Code, den Sie zur Erzeugung eigener Objekte verwenden. Hier zeigt die Variable `greeter` auf ein Objekt des Typs `Greeter`, das Sie gerade erzeugen. Wie Sie dem verschachtelten Code entnehmen können, erzeugen Sie das Objekt in zwei Schritten:

```
[[Greeter alloc] init]
```

Die Klassenmethode `alloc` ist für die Allozierung des Speichers verantwortlich und liefert ein Objekt zurück, dessen Anfangswerte durch die Instanzmethode `init` festgelegt werden. In der folgenden Zeile senden Sie die Nachricht `greeting` an `greeter` und verwenden den zurückgegebenen Wert als String, der in Ihrem Label ausgegeben wird.

4.8 Nochmalige Refaktorierung

Wir werden das nicht im gesamten Buch machen, aber ich möchte Ihnen ein Gefühl für den Rhythmus bei der Entwicklung von Cocoa-Code vermitteln. Man bekommt das Ganze nur selten beim ersten Mal richtig hin, also entwickelt man ein wenig Code und räumt dann auf, wenn einem der Code aus der Hand gleitet.

Es kommt mir ein wenig seltsam vor, dass die Methode `createLabel` für die Erzeugung des `Greeter`-Objekts verantwortlich ist. Ein möglicher Ansatz besteht darin, sich vorzustellen, wie man diese Aktion auf höherer Ebene innerhalb der Methode `applicationDidFinishLaunching:` beschreiben würde. Wir versuchen es so:

Classes/HelloWorld8/HelloWorldAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    Greeter *greeter = [self greeter];
    NSTextField * label = [self labelWithText:[greeter greeting]];
    [[self.window contentView] addSubview:label];
}
```

Bei diesem Code beziehen wir ein `Greeter`-Objekt über eine Methode namens `greeter` (die im Moment noch nicht allzu viel macht). Wir erzeugen und konfigurieren dann ein Label, indem wir dem `Greeter`-Objekt die `greeting`-Nachricht senden. Zum Schluss nehmen wir dieses Label und fügen es in den Content View des Fensters als Subview ein.

Sobald ich einen Plan dafür entworfen habe, wie meine Klasse aussehen soll, kann ich mich um die Details kümmern. So wie ich den Code aufgerufen habe, der das Label erzeugt, sind folgende Änderungen angefallen:

- Ich habe den Namen der Methode in `labelWithText:` geändert.
- Die Methode erwartet nun den Text, der im Label ausgegeben werden soll, in Form eines Parameters.
- Ich erwarte, dass die Methode das Label an mich zurückgibt, damit ich es als Subview in `applicationDidFinishLaunching:` einbinden kann, und nicht wie bisher innerhalb der Methode, die das Label erzeugt.

Die Änderungen können wir einfach an der neuen Methode `labelWithText:` vornehmen:

Classes/HelloWorld8/HelloWorldAppDelegate.m

```
► -(NSTextField *) labelWithText: (NSString *) labelText {
      NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
      NSTextField *label = [[NSTextField alloc] initWithFrame:labelFrame];
      [label setEditable:NO];
      [label setSelectable:NO];
      [label setAlignment:NSCenterTextAlignment];
      [label setFont:[NSFont boldSystemFontOfSize:36]];
►     [label setStringValue:labelText];
►     return label;
  }
```

Wir müssen außerdem eine greeter-Methode hinzufügen, die ein Greeter-Objekt erzeugt und zurückgibt:[9]

Classes/HelloWorld8/HelloWorldAppDelegate.m

```
-(Greeter *) greeter {
    return [[Greeter alloc] init];
}
```

Wir würden die Methode applicationDidFinishLaunching: gern am Anfang des Quellcodes sehen, damit man einen Blick darauf werfen und erkennen kann, was die Klasse genau tut. Wir müssen allerdings bedenken, dass der Compiler den Code von oben nach unten liest. Würde applicationDidFinishLaunching: am Anfang stehen, würde sich der Compiler darüber beschweren, dass er die greeter- bzw. die labelWithText:-Methode nicht kennt. Das Programm würde zwar trotzdem funktionieren, aber Sie würden diese Warnungen erhalten.

Wir könnten die Methoden greeter und labelWithText: in der Header-Datei deklarieren, aber sie sind nicht Teil des öffentlichen Interface. Wir werden andere Tricks kennenlernen, für den Augenblick lassen wir applicationDidFinishLaunching: aber am Schluss.

4.9 Objekte initialisieren

Wenn wir ein neues Greeter-Objekt erzeugen, allozieren wir zuerst den Speicher und rufen dann die Methode init auf. Aber Sie haben Greeter ja gesehen: Sie besitzt keine eigene init-Methode. Wenn Sie init für eine Instanz von Greeter aufrufen, rufen Sie in Wirklichkeit die init-Methode der Greeter-Superklasse NSObject auf.

Diese Vererbung funktioniert gut, weil Greeter keine eigenen Variablen initialisieren muss. In Greeter war nichts enthalten, das es nicht bereits in NSObject gibt und eine Initialisierung verlangt.

Das wollen wir ändern. Lassen Sie uns Greeter um eine Instanzvariable namens name erweitern. Denken Sie daran, dass eine Deklaration in der Header-Datei Greeter.h zwischen den geschweiften Klammern stehen muss.

9 Wir hätten die Erzeugung des Greeter-Objekts auch in der Methode application-DidFinishLaunching: lassen können. Ich habe sie hier herausgezogen, um kleine Methoden zu betonen und Ihnen die Möglichkeit zu geben, zusätzliche Erfahrungen mit der Entwicklung von Methoden und dem Senden von Nachrichten zu sammeln.

Classes/HelloWorld9/Greeter.h

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

@interface Greeter : NSObject {
►     NSString * name;
}
-(NSString *) greeting;
@end
```

Was stellen wir nun mit dieser neuen Variablen an? Zuerst müssen wir ihren Wert festlegen, wenn das Objekt initialisiert wird. Wir schreiben unsere eigene `init`-Methode, mit der wir den Wert von `name` auf *World* setzen. Dann müssen wir die Implementierung von `greeting` so abändern, dass es den Wert von `name` zwischen *Hello* und *!* einfügt.

Wir gehen den Code in Ruhe durch und beginnen mit `init`:

Classes/HelloWorld9/Greeter.m

```
1  #import "Greeter.h"
2
3  @implementation Greeter
4
5  -(NSString *) greeting {
6      return [[NSString alloc] initWithFormat:@"Hello, %@!", name];
7  }
8
9  -(id) init {
10     if (self =[super init]){
11         name = @"World";
12     }
13     return self;
14 }
15 @end
```

Beginnen wir mit Zeile 10. Die sieht irgendwie nicht richtig aus. Sie könnten denken, dass hier zwei Gleichheitszeichen zwischen `self` und `[super init]` stehen sollten, aber tatsächlich handelt es sich hier um eine Zuweisung und nicht um einen Test auf Gleichheit. Was passiert also bei dieser Zuweisung?

```
self =[super init];
```

Das Erste, was Sie bei der Initialisierung Ihres Objekts machen, ist der Aufruf der `init`-Methode der Superklasse. Sie weisen das initialisierte Objekt `self` zu und stellen dann sicher, dass `self` nicht `nil` ist. Ist das nicht der Fall, machen Sie weiter und initialisieren die Variablen wie in Zeile 11. Auf diese Weise können Sie alle Initialisierungen vornehmen, die es vor dieser Methode gab, und darüber dann die zusätzliche Initia-

lisierung für diese Klasse legen. Zum Schluss geben Sie Ihr Objekt in Zeile 13 zurück.

Häufig werden Sie der folgenden, etwas weitschweifigeren, aber gleichwertigen Variante begegnen:

```
-(id) init
{
    self =[super init];
    if (self != nil) {
        name = @"World";
    }
return self;
}
```

Wenn Sie diese Version verwenden, würde ich empfehlen, `if (self != nil)` durch `if(self)` zu ersetzen.[10]

Sehen wir uns nun die sechste Zeile der `greeting`-Methode an. Wie geben einen formatierten String zurück. Das entspricht dem, was Sie aus vielen anderen Programmiersprachen kennen. Sie verwenden einen String, der Platzhalter wie `%d` enthält. Diesem String folgt eine Liste kommaseparierter Werte, die die Platzhalter in diesem String ersetzen.

In unserem Beispiel nutzen wir den neuen Cocoa-Typ `%@`, der die String-Repräsentation eines Cocoa-Objekts anzeigt (wie in seiner `description`-Methode definiert). In unserem Fall verwenden wir das hier:[11]

```
[[NSString alloc] initWithFormat:@"Hello, %@!", name]
```

Wir ersetzen also das `%@` durch den Wert von `name`, der im Moment auf `World` gesetzt ist.[12] Das Ergebnis ist `Hello, World!`.

Klicken Sie auf *Build & Run*, und das Programm sollte wieder genau arbeiten wie vorher.

---

10 Eine vollständige Betrachtung von `self` und `[super init]` finden Sie in Matt Gallaghers Artikel auf *http://cocoawithlove.com/2009/04/what-does-it-mean-when-you-assign-super. html*. Informativ und unterhaltsam ist auch Wil Shipleys Artikel zur Verwendung von `self= [super init]`, den Sie auf *http://www.wilshipley.com/blog/2005/07/self-stupid-init.html* finden. Lesen Sie ihn bis zu Schluss: Da folgert er, dass es eine gute Sache ist, und erläutert, wie man die Methode nutzen kann. Sie finden dort auch einen Verweis auf eine umfassende Liste der Stringformat-Specifier.

11 Wenn Sie das Speichermanagement von Objective-C bereits kennen, merken Sie, dass ich hier ein *Speicherleck* eingeführt habe. Ich habe das bewusst getan, und es wird vom Garbage Collector abgefangen. Wir werden später darüber reden, was zu tun ist, wenn kein Garbage Collector zur Verfügung steht.

12 Details zu Formatstrings finden Sie in der Dokumentation zu `stringWithFormat:`. Sie verweist auf einen Artikel namens „Formatting String Objects“, der einige Beispiele enthält.

4.10 Logging von Objekten

Manchmal möchte man an bestimmten Stellen eines Programms einen kurzen Blick auf den Zustand eines Objekts werfen. Eine schnelle Lösung bietet die Verwendung von `NSLog()` zusammen mit `%@`, um die Stringbeschreibung des Objekts auszugeben. Leider kann Apple nicht wissen, was Sie für Ihr Objekt festhalten wollen. Fügen Sie zum Beispiel die hervorgehobene Zeile ein, um den Wert des neu erzeugten `Greeter`-Objekts auszugeben:

Classes/HelloWorld10/HelloWorldAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    Greeter *greeter = [self greeter];
►   NSLog(@"Greeter: %@", greeter);
►   NSLog(@"This occurred in %@ at line %d in file %s.",
►                       NSStringFromSelector(_cmd), __LINE__, __FILE__);
    NSTextField * label = [self labelWithText:[greeter greeting]];
    [[self.window contentView] addSubview:label];
}
```

Klicken Sie auf *Build & Run*. Leider weiß das `Greeter`-Objekt nicht, wie es sich selbst beschreiben soll, Sie erben also das Standardverhalten von `NSObject`, und Sie sehen so etwas wie:

```
Greeter <Greeter: 0x10044d520>
```

Das ist nur der Klassenname des Objekts samt seiner Speicherposition. Das ist nicht besonders nützlich, aber Sie können immer genau festlegen, wie sich ein Objekt darstellt, indem Sie seine `description`-Methode überschreiben. Um das Debugging zu unterstützen, gebe ich den Namen der Methode sowie den Dateinamen und die aktuelle Zeilennummer aus.[13]

Classes/HelloWorld10/Greeter.m

```
-(NSString *)description {
    return [[NSString alloc] initWithFormat:@"name: %@ \n created: %@",
            name, [NSDate date]]; }
```

Nun weiß `Greeter`, wie es sich selbst beschreiben kann. Wenn Sie die Anwendung erneut ausführen, erhalten Sie so etwas:

```
Greeter: name: World
 created: 2010-02-10 15:57:35 -0500
This occurred in applicationDidFinishLaunching: at line 27 in file
/Volumes/Data/Prags/Bookshelf/Writing/DSCPQSL/Book/code/Classes
/HelloWorld10/HelloWorldAppDelegate.m.
```

13 Weitere Optionen finden Sie in Apples „Improved logging in Objective C“ unter *http://developer.apple.com/mac/library/qa/qa2009/qa1669.html*.

Es gibt keine festen Vorgaben für die Implementierung der `description`-Methode. Nehmen Sie also alle Daten auf, die Sie für sich und andere als sinnvoll erachten. Sie sehen in dieser Logmeldung, wie Apple die `description`-Methode für die Klasse `NSDate` implementiert hat. Die Ausgabe besteht aus Jahr, Monat, Tag, Uhrzeit und Zeitzone.

4.11 Übung: Zusätzliche Initialisierung

Wie es bei diesem Beispiel immer so ist, werden Sie nach einer gewissen Zeit nicht immer die ganze Welt, sondern auch mal jemand anderen grüßen wollen. Glücklicherweise sind wir darauf bestens vorbereitet. Um jeden Grüßen zu können, den wir grüßen wollen, müssen wir in der Lage sein, die Variable `name` zurückzusetzen.

Eine Möglichkeit, um die Flexibilität etwas zu erhöhen, besteht darin, eine zweite `init`-Methode zu entwickeln. Diese würde dann den Namen als Parameter übergeben. Die Konvention lautet, einen etwas anschaulicheren, mit *init* beginnenden Methodennamen zu verwenden. Wir könnten sie `initWithName:` nennen.

Nehmen Sie sich eine Minute Zeit und deklarieren Sie in der Header-Datei eine `initWithName:`-Methode, die einen `NSString *` als einzigen Parameter verlangt und eine `id` zurückliefert. In `Greeter.m` implementieren Sie dann die `initWithName:`-Methode, die den Wert von `name` auf den übergebenen String festlegt.[14]

Jede Klasse sollte ein designiertes `init` besitzen, das alle anderen `inits` aufruft. Refaktorieren wir unsere aktuelle `init`-Methode so, dass sie `initWithName:` aufruft und dabei `World` als Parameter übergibt.

Klicken Sie auf *Build & Run*. Unsere Anwendung sollte wie gewohnt ausgeführt werden.

In `HelloWorldAppDelegate.m` benennen Sie die `greeter`-Methode in `greeterFor` um. Diese verlangt einen `NSString *` als einziges Argument und erzeugt einen `Greeter` mithilfe der `Greeter`-Methode `initWithName:`.

Klicken Sie auf *Build & Run*. Sie sollten eine etwas persönlichere Begrüßung sehen.

14 Ich habe bisher noch nichts zu `id` gesagt. Stellen Sie es sich als einen Typ vor, der für jeden möglichen Objekttyp steht. Ich werde in Kapitel 5, *Instanzvariablen und Eigenschaften*, auf Seite 77 mehr darüber verraten.

4.12 Lösung: Zusätzliche Initialisierung

Wir nehmen die folgende Deklaration in die `Greeter`-Header-Datei auf, weil wir das Public Interface um eine neue Initialisierungsmethode erweitern:

Classes/HelloWorld11/Greeter.h

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

@interface Greeter : NSObject {
    NSString * name;
}
-(NSString *) greeting;
► -(id) initWithName:(NSString *)newName;
@end
```

In `Greeter.m` nehmen wir diese Methode auf:

Classes/HelloWorld11/Greeter.m

```
-(id) initWithName:(NSString *) newName {
    if (self =[super init]){
        name = newName;
    }
    return self;
}
```

Sobald Sie im nächsten Kapitel etwas über Eigenschaften (properties) gelernt haben, werden Sie etwas Robusteres verwenden als

```
name = newName;
```

Sie werden nämlich eine Eigenschaft nutzen, um den Wert von `name` festzulegen und sich dabei auch um das Speichermanagement kümmern.

Sie können die `init`-Methode nun so refaktorieren, dass sie `initWithName:` aufruft. Auf diese Weise eliminieren wir den doppelten Code.

Classes/HelloWorld11/Greeter.m

```
-(id) init {
    return [self initWithName:@"World"];
}
```

Diese Methode ersetzt `greeter` in `HelloWorldAppDelegate.m`:

Classes/HelloWorld11/HelloWorldAppDelegate.m

```
-(Greeter *) greeterFor:(NSString *) personName {
    return [[Greeter alloc] initWithName:personName];
}
```

Passen Sie `applicationDidFinishLaunching:` so an, dass es die neue Methode aufruft:

Classes/HelloWorld11/HelloWorldAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
►    Greeter *greeter = [self greeterFor:@"Maggie"];
     NSLog(@"Greeter: %@", greeter);
     NSTextField * label = [self labelWithText:[greeter greeting]];
      [[self.window contentView] addSubview:label];
}
```

Klicken Sie auf *Build & Run*, und es erscheint Folgendes:

In diesem Kapitel haben Sie eigene Klassen und Objekte erzeugt und eingesetzt. Sie haben eigene Initialisierer, Klassenmethoden, Instanzmethoden und Instanzvariablen entwickelt. Nun gehen wir einen Schritt zurück und sehen uns einige der Aspekte an, die wir unter den Teppich gekehrt haben.

Kapitel 5

Instanzvariablen und Eigenschaften

Nachrichten sind in diesem Buch ein wichtiges Thema. Bisher haben wir Nachrichten an ein Objekt gesendet, damit es Hallo sagt oder sich mit einem bestimmten Namen initialisiert. In diesem Kapitel werden wir uns Nachrichten ansehen, die den Zustand der Variablen eines Objekts abrufen oder festlegen.

Viele Ihrer Objekte besitzen Instanzvariablen zur Speicherung von Zuständen. Häufig ist dieser Zustand objektintern – Sie möchten nicht, dass er anderen Objekten zugänglich ist. Aber manchmal sollen andere Objekte in der Lage sein, den Wert einer oder mehrerer Instanzvariablen zu untersuchen und eventuell auch zu verändern. Denken Sie an einige der Variablen zurück, die wir im Interface Builder mithilfe des Attributes Inspector gesetzt haben. Wir haben den Text für die Buttons sowie die entsprechenden Tastaturkürzel festgelegt. Im vorigen Kapitel haben wir bestimmte Attribute wie die Schriftgröße und die Ausrichtung des Textfeldes festgelegt. Diese öffentlich sichtbaren Instanzvariablen sind *Eigenschaften* (properties) des Objekts. Objective-C 2.0 besitzt einen Mechanismus zur Generierung der sogenannten *Getter* und *Setter*, über die andere Objekte auf sie zugreifen können.

Wir wollen mit Instanzvariablen beginnen und diese über selbstdeklarierte und -entwickelte Getter und Setter bereitstellen. Wir generieren die Deklaration dann automatisch mithilfe der `@property`-Direktive und implementieren sie automatisch über die `@synthesize`-Direktive. Wir werden dann lernen, wie man diese Akzessoren über die neue Punktsyntax aufruft.

5.1 Zeiger

Sehen wir uns die Header-Datei für `Greeter` noch einmal an:

Properties/HelloWorld12/Greeter.h

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

@interface Greeter : NSObject {
    NSString *name;
}
-(NSString *) greeting;
-(id) initWithName:(NSString *)name;
@end
```

Jedes Mal, wenn Sie den Typ `NSString` sehen, folgen diesem ein Leerzeichen und ein Sternchen (Asterisk, `*`). Betrachten Sie insbesondere die Deklaration der Instanzvariablen:

```
NSString *name;
```

Bei Objective-C sind auf Objekte verweisende Variablen in Wirklichkeit Zeiger. Das `*` weist darauf hin, dass `name` ein Zeiger ist. Wir können unsere Variable auf zwei Arten deklarieren:

```
NSString* name; // OK, aber
NSString *name; // diese Form wird bevorzugt
```

Bei Objective-C neigen wir dazu, das `*` direkt vor den Variablennamen zu setzen.

Wir achten sorgfältig darauf, eine Objektvariable mit `*` als Zeiger zu kennzeichnen, sind allerdings etwas salopper, wenn wir über sie reden. Wir sagen beispielsweise, dass `name` ein `NSString` ist, obwohl `name` eigentlich ein Zeiger auf einen `NSString` ist.[1]

Wir verwenden Zeiger auch in Methodendeklarationen, hier zum Beispiel in der Deklaration einer Instanzmethode namens `greeting`, die einen Zeiger auf einen `NSString` zurückgibt:

```
-(NSString *) greeting;
```

Auch hier werden wir in einer normalen Unterhaltung eher davon sprechen, dass `greeting` einen String zurückgibt, auch wenn der Code deutlich macht, dass wir in Wirklichkeit einen Zeiger auf einen `NSString` zurückliefern.

---

1 In der Java-Welt gilt das Gleiche, aber man tut so, als gäbe es bei Java keine Zeiger: Man gerät also in Erklärungsnot, wenn die erste `NullPointerException` ausgelöst wird.

Sehen wir uns zum Schluss noch die Deklaration unserer benutzerdefinierten `init`-Methode an:

```
-(id) initWithName:(NSString *)newName;
```

Hier deklarieren wir eine Instanzmethode namens `initWithName:`. Diese Methode verlangt einen Zeiger auf einen `NSString` namens `newName` als einziges Argument.

Der Rückgabetyp für `initWithName:` ist `id`.Wir verwenden `id` für einen Zeiger, dessen Typ wir nicht exakt angeben wollen. Mit anderen Worten nutzen wir `id` in Objective-C, um auf die Instanz einer Klasse zu verweisen. Wir haben `id` im Abschnitt 3.3, *Methoden mit Argumenten*, auf Seite 43 als Parametertyp genutzt.

```
-(IBAction) goBack:(id) sender
```

Das ist die Nachricht, die an den `WebView` gesendet wird, wenn man den *Back*-Button anklickt.[2] So wie wir es verknüpft haben, war der `sender` der Nachricht ein `NSButton *`. Wir hätten ein ganz anderes Element einführen und zur Initiierung der `goBack:`-Nachricht verwenden können, der `sender`-Typ wäre dann ein völlig anderer gewesen. Wir bewahren uns die nötige Flexibilität, indem wir den `sender`-Typ mit `id` festlegen.

Er wird nicht `id *` geschrieben, weil `id` schon ein Zeiger ist. Das kann ein wenig verwirrend sein, weil `id` für Zeiger wie `NSButton *` und `NSString *` steht. Doch `id` wird ohne das Sternchen geschrieben.

In unserem Beispiel haben wir `id` als Rückgabetyp für jedes `init` (einschließlich unseres eigenen `initWithName:` und anderer) verwendet, so wie es in Apples *The Objective-C Programming Language* [App09f] beschrieben wird.

5.2 Mit Nicht-Objekten arbeiten

Objective-C setzt auf C auf, Sie können also primitive C-Typen wie `int`, `long`, `float` und so weiter verwenden. Ihnen werden aber auch andere Typen begegnen, die weder Primitive noch Objekte sind.

2 Wir sehen uns den Rückgabetyp `IBAction` in Kapitel 7, *Outlets und Aktionen*, auf Seite 117 an. Eine `IBAction` entspricht einem `void`, nur dass der Interface Builder uns einige Informationen dazu liefern kann. Sehen Sie sich die Dokumentation an: Sie werden sehen, dass `IBAction` als `void` definiert ist.

Wenn Sie sich zum Beispiel die erste Zeile der `labelWithText:`-Methode in `HelloWorldAppDelegate.m` ansehen, erkennen Sie Folgendes:

```
NSRect labelFrame = NSMakeRect(20, 20, 540, 50);
```

Als Sie das eingegeben haben, werden Sie nicht groß darüber nachgedacht haben, aber jetzt sieht es so aus, als würde das * fehlen. Das ist nicht der Fall. Wenn Sie die Dokumentation bemühen, werden Sie sehen, dass `NSRect` ein Struct darstellt, das aus einem `NSPoint` und einer `NSSize` besteht. Diese sind wiederum als Structs aus zwei `CGFloats` definiert. Geht man in der Dokumentation noch einen Schritt weiter, sieht man, dass `CGFloat` per *typedef* als `float` (für 32-Bit-Code) bzw. als `double` (für 64-Bit-Code) definiert ist. Eine Variable des Typs `NSRect` ist also kein Zeiger, und daher gibt es auch kein *. [3]

Das kann recht verwirrend sein. `NSNumber` ist ein Objekt, `NSInteger` ist keins. Sie müssen in der Dokumentation nachsehen, was was ist. Es folgt die Deklaration zweier Variablen, einer vom Typ `NSNumber` und einer vom Typ `NSInteger`:

```
NSNumber *einkommen;
NSInteger alter;
```

Objective-C kennt außerdem den Typ `BOOL` für boolesche Werte. Wir wollen komplizierte und unlesbare Arithmetik bei logischen Berechnungen vermeiden. Daher besitzt ein `BOOL` nur die Werte `YES` oder `NO`. Wir achten insbesondere darauf, im Zusammenhang mit booleschen Werten nicht Wörter wie *true* (wahr) oder *false* (falsch) zu benutzen. Wir wollen ja nicht, dass die Leute denken, wir wären nicht von hier.

5.3 Getter und Setter

Wir können den Wert von `name` innerhalb der `Greeter`-Klasse leicht abrufen (get) und setzen (set). In der ersten hervorgehobenen Zeile des folgenden Beispiels rufen wir den Wert von `name` ab und geben ihn aus, während wir in der zweiten hervorgehobenen Zeile seinen Wert setzen:

3 Eine sehr gute Erläuterung von Objekten und einfachen C-Strukturen finden Sie in der Antwort zu einer Stacküberlauf-Frage unter *http://stackoverow.com/questions/2189212/why-object-dosomething-and-not-object-dosomething*. Die Antwort wurde von Apple-Ingenieur Bill Bumgarner geschrieben, der üblicherweise unter dem Namen *bbum* postet. Es lohnt sich immer, seine Postings zu Stacküberläufen und seine Beiträge in verschiedenen anderen Listen (wie *cocoa-dev*) auf *http://lists.apple.comhttp://lists.apple.com* anzusehen. Von Zeit zu Zeit fasst er seine Gedanken in seinem Blog unter *http://www.friday.com/bbum/http://www.friday.com/bbum/* zusammen.

Properties/HelloWorld12/Greeter.m

```
#import "Greeter.h"

@implementation Greeter
-(NSString *) greeting {
    return [[NSString alloc] initWithFormat:@"Hello, %@!", name];
}
-(id) initWithName:(NSString *) newName {
    if (self =[super init]){
        name = newName;
    }
    return self;
}
-(id) init {
    return [self initWithName:@"World"];
}
-(NSString *)description {
    return [[NSString alloc] initWithFormat:@"name: %@ \n created: %@",
            name, [NSDate date]]; }
@end
```

Joe fragt...

Muss ich diesen Namenskonventionen folgen?

Ja. Es ist wichtig, diesen Namenskonventionen zu folgen, weil sie für viele der Zaubereien nötig sind, deren Zeuge Sie in Kürze sein werden. Es handelt sich nur um Koventionen, der Compiler zwingt Sie also nicht dazu, diese Namen zu verwenden – aber es ist sehr wichtig, dass Sie die Konventionen befolgen.

Sollen andere Objekte den Wert von `name` abrufen und setzen können, deklarieren wir einen Getter und einen Setter in der `Greeter`-Header-Datei und implementieren sie in der Implementierungsdatei.

Der Getter ist eine Instanzmethode, die den aktuellen Wert der Instanzvariablen `name` zurückgibt, der Rückgabewert muss also vom Typ `NSString *` sein. Der Getter arbeitet ohne Parameter. Die einzige Frage, die Sie sich stellen müssen, ist die nach dem richtigen Namen. In Objective-C hat der Getter für die Eigenschaft `xyz` den Namen `xyz` – ein *get* kommt im Namen gar nicht vor.

Der Setter ist ebenfalls eine Instanzmethode, verlangt aber den neuen Wert von `name` als `NSString *`, den wir `name` nennen werden. Was den Methodennamen betrifft, wird beiObjective-C der Setter für die Eigen-

schaft xyz mit setXyz benannt. Mit anderen Worten beginnen Sie den Namen mit dem Wort *set*, dann kommt der Variablenname, dessen erster Buchstabe großgeschrieben wird.

Fügen Sie die Deklarationen für alle Akzessoren in die Header-Datei ein:

Properties/HelloWorld13/Greeter.h

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

@interface Greeter : NSObject {
     NSString *name;
}
-(NSString *) greeting;
-(id) initWithName:(NSString *)name;
-(NSString *) name;
-(void) setName:(NSString *) name;

@end
```

Zum Getter gibt es nicht viel zu sagen. Solchen Code können Sie wahrscheinlich im Schlaf schreiben:

Properties/HelloWorld13/Greeter.m

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

Wenn wir das Speichermanagement für den Augenblick ignorieren, ist der Setter kaum mehr als das hier:

Properties/HelloWorld13/Greeter.m

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

Wir können nun die direkten Aufrufe von Instanzvariablen in Aufrufe der Getter und Setter umwandeln. Beim Aufbau des formatierten Strings würden wir den Getter beispielsweise so aufrufen:

Properties/HelloWorld13/Greeter.m

```
-(NSString *) greeting {
    return [[NSString alloc] initWithFormat:@"Hello, %@!",[self name]];
}
```

Und in der initWithName:-Methode würden wir den Setter so aufrufen:

Properties/HelloWorld13/Greeter.m

```
-(id) initWithName:(NSString *) newName {
    if (self =[super init]){
        [self setName: newName];
```

```
    }
    return self;
}
```

Noch wichtiger ist aber, dass wir die `name`-Eigenschaft nun aus anderen Objekten abrufen können. Hier ein (etwas albernes) Beispiel, das ich in `HelloWorldAppDelegate.m` aufgenommen habe:

Properties/HelloWorld13/HelloWorldAppDelegate.m

```
-(void) setUpperCaseName:(Greeter *) greeter {
    NSLog(@"The name was originally %@.", [greeter name]);
    [greeter setName:[[greeter name] uppercaseString]];
    NSLog(@"The name is now %@.", [greeter name]);
}
```

Ich habe in jeder Zeile den Getter `[greeter name]` benutzt. Der Setter `setName:` wird in der mittleren Zeile verwendet, um den Wert von `name` in Großbuchstaben umzuwandeln.

Damit das funktioniert, müssen Sie einen Aufruf dieser Methode in `applicationDidFinishLaunching:` einfügen, so wie es hier zu sehen ist:

Properties/HelloWorld13/HelloWorldAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
     Greeter *greeter = [self greeterFor:@"Maggie"];
         NSLog(@"Greeter: %@", greeter);
     NSTextField * label = [self labelWithText:[greeter greeting]];
►    [self setUpperCaseName:greeter];
    [[self.window contentView] addSubview:label];
}
```

Klicken Sie auf *Build & Run*. Sie sehen die folgende Ausgabe:

```
The name was originally Maggie.
The name is now MAGGIE.
```

Nun wollen wir Eigenschaften einführen und damit beginnen, unseren Code zu reduzieren und zu vereinfachen.

5.4 Akzessoren in Eigenschaften umwandeln

Fassen wir zusammen, wie weit wir gekommen sind. Wir haben ein Attribut der `Greeter`-Klasse namens `name` offengelegt, das wir abrufen und setzen können. Wir haben das in drei Schritten getan:

1. Deklaration der Instanzvariablen `name` in der Header-Datei.
2. Deklaration eines Getters und eines Setters namens `name` und `setName:` in der Header-Datei.

3. Implementierung der Methoden in der Implementierungsdatei durch Standardcode.

Nun wollen wir die Eigenschaften von Objective-C 2.0 verwenden, um Änderungen an diesen letzten beiden Schritten vorzunehmen und in manchen Fällen den ersten Schritt zu eliminieren.

Ersetzen Sie die Deklarationen der Getter und Setter so:

Properties/HelloWorld14/Greeter.h

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

@interface Greeter : NSObject {
    NSString *name;
}
-(NSString *) greeting;
-(id) initWithName:(NSString *)name;
► @property(copy) NSString *name;

@end
```

Ignorieren Sie für den Moment das in Klammern stehende `copy` hinter der Compilerdirektive `@property`. Interpretieren Sie den Rest der Zeile als Deklaration einer Eigenschaft vom Typ `NSString` mit dem Namen `name`. So wie sie da steht, deklariert die Zeile einen Getter namens `name` und einen Setter namens `setName:`.

Wenn Sie jetzt *Build & Run* anklicken, funktioniert Ihr Code genau wie vorher: Sie können ohne irgendwelche Warnungen oder Fehlermeldungen den Getter und Setter aus `HelloWorldAppDelegate` aufrufen. Mit anderen Worten deklariert diese `@property`-Direktive Ihre Getter und Setter; eigentlich tut sie noch mehr als das, wie Sie im Abschnitt 5.6, *Eigenschaftsattribute*, auf Seite 88 noch sehen werden.

An unserer Implementierung der Getter- und Setter-Methoden war nichts Besonderes dran. Tatsächlich kann der Compiler diese Implementierungen für uns erzeugen, wenn wir ihn darum bitten. Ersetzen Sie Ihre Implementierung des Getters und des Setters durch die hervorgehobene Zeile:

Properties/HelloWorld14/Greeter.m

```
#import "Greeter.h"

@implementation Greeter

► @synthesize name; ##hervorheben! (hier und an weiteren Stellen)##
```

Die Direktive `@synthesize` weist den Compiler an, die Methoden zu implementieren, die in der entsprechenden `@property`-Direktive deklariert wurden.

Sehen wir uns unsere drei Schritte zur Entwicklung von Eigenschaften also noch einmal an. Wir deklarieren eine Instanzvariable in der Header-Datei zwischen den geschweiften Klammern. Wir deklarieren die Getter und Setter in der Header-Datei außerhalb der geschweiften Klammern mithilfe der `@property`-Direktive. Bisher deklarieren wir nur eine Variable und eine Eigenschaft (mit anderen Worten die Akzessormethoden), und das geschieht ja bekanntlich in der Header-Datei. Nun wird es Zeit, die Methoden zu implementieren, die wir für die Variable deklariert haben. Implementierungen stehen in der `.m`-Datei, weshalb wir dort die `@synthesize`-Direktive eintragen.

Klicken Sie *Build & Run* an, und das Programm läuft genau wie vorher. Mit anderen Worten wurden die Getter und Setter korrekt generiert.

5.5 Punktnotation

Nachdem Sie die Eigenschaften deklariert und synthetisiert haben, sollten Sie sie anders aufrufen. Wenn Sie einen Getter benötigen, sollten Sie anstelle der bisher verwendeten Methodenaufrufsyntax

```
[self name]
```

die Punktnotation verwenden:

```
self.name
```

Konkret ändern Sie die hervorgehobene Zeile der `greeting`-Methode wie folgt:

Properties/HelloWorld14/Greeter.m

```
-(NSString *) greeting {
►    return [[NSString alloc] initWithFormat:@"Hello, %@!", self.name];
}
```

In ähnlicher Weise ersetzen Sie die Setter-Version

```
[self setName:newName];
```

durch diese:

```
self.name = newName;
```

Die `initWithName:`-Methode ändert sich wie folgt:

Properties/HelloWorld14/Greeter.m

```
-(id) initWithName:(NSString *) newName {
    if (self =[super init]){
►        self.name = newName;
    }
    return self;
}
```

In jedem dieser Fälle verhalten sich der Methodenaufruf und die Punktsyntax identisch. Tatsächlich ist die Punktsyntax nur ein syntaktisches Schmankerl, das vom Compiler in einen Methodenaufruf umgewandelt wird. Wenn Sie aber konsequent die Punktsyntax zum Abrufen und Setzen von Eigenschaften nutzen, werden Sie feststellen, dass Ihr Code einfacher zu lesen ist.

Joe fragt...

Warum sollte ich mir um die Punktsyntax Gedanken mach?

Dieses Thema wird kontrovers diskutiert. Einige Leute weisen darauf hin, dass bei Code wie

```
self.name = newName;
```

eine Methode aufgerufen wird, auch wenn es nicht nach einem Methodenaufruf aussieht. Das ist wahr. Aber der Vorteil der Punktsyntax liegt darin, dass Abrufen und Setzen einer Eigenschaft eine andere Aktivität darstellen als das Senden einer Nachricht an ein Objekt, selbst wenn der darunterliegende Mechanismus derselbe ist. Die Punktsyntax macht Ihre Absicht deutlich.

Das Senden einer Nachricht wie

```
[greeter greeting];
```

ist offensichtlich etwas völlig anderes als das Abrufen oder Setzen des Wertes einer Eigenschaft. Sie werden feststellen, dass Apple die Punktsyntax in seinen modernen APIs bevorzugt. Wenn Sie sich die Aufgaben einer Klasse ansehen, werden Sie feststellen, dass es sich bei vielen tatsächlich um Eigenschaften handelt. Nachdem Sie die Punktnotation eine Weile zum Abrufen und Setzen benutzt haben, werden Sie feststellen, dass Ihr Code einfacher zu lesen und zu verstehen ist.

Auf *http://eschatologist.net/blog/?p=160* erläutert Chris Hanson leidenschaftlich und detailliert, warum Sie die Punktsyntax für Eigenschaften nutzen sollten.

Das fühlt sich an, als drehten wir uns im Kreis. Wenn wir uns beispielsweise den Setter ansehen, haben wir zuerst damit begonnen, die Variable direkt zu setzen:

```
name = newName;
```

Dann haben wir den Setter eingeführt. Man hat Ihnen gesagt, Sie sollen die Setter-Methode verwenden, um den Wert der zugrunde liegenden Instanzvariablen festzulegen:

```
[self setName:newName];
```

Jetzt haben Sie eine Eigenschaft deklariert und synthetisiert und sollen der Wert der Variablen nun wie folgt setzen:

```
self.name = newName;
```

Nein, wir drehen uns nicht im Kreis. Wenn Sie `self.name = newName` verwenden, nutzen Sie die Setter-Methode und greifen nicht direkt auf die darunterliegende Variable zu.

Sie sollten nicht die Variable, sondern die Eigenschaft verwenden, um die Vorteile der verschiedenen Einstellungen zu nutzen, um die wir unsere Eigenschaft gleich erweitern werden. Das ist einer der häufigsten Fehler im Zusammenhang mit Eigenschaften: Sie nehmen den ganzen Ärger der Deklaration und Synthetisierung einer Eigenschaft auf sich, vergessen dann aber, mit `self` zu arbeiten und verwenden die Instanzvariable direkt, statt die Eigenschaft zu verwenden.

Noch wichtiger ist, dass wir auf diese Eigenschaft aus `HelloWorldAppDelegate` zugreifen können. Zuerst möchte ich, dass Sie sich `HelloWorldApp-Delegate` noch einmal ansehen:

Properties/HelloWorld14/HelloWorldAppDelegate.h

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

@interface HelloWorldAppDelegate : NSObject <NSApplicationDelegate> {
     NSWindow *window;
}
@property (assign) IBOutlet NSWindow *window;
@end
```

Nachdem Sie nun über Eigenschaften Bescheid wissen, können Sie sehen, dass die Variable `window` vom Typ `NSWindow` als Eigenschaft deklariert ist. Wenn Sie sich den Anfang von `HelloWorldAppDelegate.m` ansehen, bemerken Sie, das sie synthetisiert wurde. Und vielleicht erinnern Sie sich daran, dass wir am Ende von `applicationDidFinish-Launching:` auf diese Eigenschaft zugegriffen haben, als wir das Label in den Content-View eingefügt haben:

```
[[self.window contentView] addSubview:label];
```

Sobald Sie den Code in der `setUpperCaseName:`-Methode so refaktoriert haben, dass er die Punktsyntax nutzt, sollte sie wie folgt aussehen:

Properties/HelloWorld14/HelloWorldAppDelegate.m

```
-(void) setUpperCaseName:(Greeter *) greeter {
    NSLog(@"The name was originally %@.", greeter.name);
    greeter.name = [greeter.name uppercaseString];
    NSLog(@"The name is now %@.", greeter.name);
}
```

5.6 Eigenschaftsattribute

Die allgemeine Form der Deklaration einer Eigenschaft sieht so aus:

```
@property(attribut1, attribut2,...) PropertyTyp propertyName;
```

Die Attribute legen fest, welche Art von Akzessoren bei der Synthese erzeugt werden sollen.

Ein Satz von Attributen legt fest, ob Sie einen Getter und einen Setter oder einfach nur einen Getter erzeugen wollen. Standardmäßig werden beide generiert, Sie müssen das Attribut `readwrite` also nicht verwenden; wenn aber nur ein Getter erzeugt werden soll, verwenden Sie `readonly`.

Ein anderer Satz von Attributen ist beim Setzen der Akzessornamen hilfreich. Das wird häufig bei booleschen Werten genutzt. Sie könnten beispielsweise eine Variable namens `highlighted` verwenden. Der Standardname für den Getter würde `highlighted` lauten, doch der natürlichere Name für ein `BOOL` wäre `isHighlighted`. Sie ändern den Namen der generierten Akzessoren über die `getter`- und `setter`-Attribute:

```
@property(getter=isHighlighted) BOOL highlighted;
```

Bei booleschen Eigenschaften müssen Sie diese Änderung vornehmen, um den vom System verwendeten Namenskonventionen zu folgen, die einen Teil der Dynamik ermöglichen, die Sie im Verlauf des Buches noch kennenlernen werden.

Wir haben noch nicht über das Speichermanagement gesprochen, weshalb es etwas schwierig ist, die Optionen des Speicherattributs zu erläutern. Die folgenden Zeilen sollen Ihnen darüber hinweghelfen, bis wir zu Seite 95 in Kapitel 6, *Speicher*, kommen. Die Möglichkeiten sind `assign` (was der Voreinstellung entspricht), `retain` und `copy`.

Hier kommen die grundsätzlichen Unterschiede. Sie verwenden `assign` für alle Nicht-Objekte. Stellen Sie sich eine Eigenschaft vom `NSInteger` namens `age` vor. Wenn Sie `assign` wählen, sieht der Setter wie folgt aus:

```
-(void) setAge:newAge {
      age = newAge;
} //assign
```

Bei Objekttypen arbeitet man mit Zeigern. Zuerst bestimmen wir, ob der alte und der neue Zeiger auf dasselbe Objekt verweisen. Ist das nicht der Fall, müssen wir (wie Sie im nächsten Kapitel erfahren werden) das Laufzeitsystem wissen lassen, dass wir nicht länger an dem Objekt interessiert sind, auf das die Variable verweist. Wir machen das, indem wir die Nachricht `release` an unsere Variable senden.

Nun haben wir zwei Möglichkeiten. Wenn die Variable Kopien unterstützt, können wir eine lokale Kopie des Objekts erzeugen und mit dieser Kopie arbeiten, ohne das übergebene Objekt zu verändern. Formal ausgedrückt, können wir `copy` nur verwenden, wenn die Eigenschaft das `NSCopying`-Protokoll unterstützt. Für `NSString` trifft das zu,[4] wenn wir also das `copy`-Attribut nutzen, sieht unser Setter wie folgt aus:

```
-(void) setName:newName {
     if (name != newName) {
         [name release];
         name = [newName copy];
     }
} //copy
```

Die andere Option besteht darin, die der Eigenschaft zugrunde liegende Variable auf das Objekt verweisen zu lassen, auf das das Argument zeigt. Das ist die Zeigervariante von `assign`, und wenn es nicht die Speicherverwaltung gäbe, wäre die Implementierung genau gleich. Nehmen wir eine Eigenschaft namens `buddy` vom Typ `Greeter`. Der Setter sieht etwa so aus:

```
-(void) setBuddy:newBuddy {
     if (buddy != newBuddy) {
         [buddy release];
         buddy = [newBuddy retain];
     }
} //retain
```

Diese Regeln gelten sowohl für die iPhone- als auch für die Mac OS X-Entwicklung. Wenn Sie den Garbage Collector nutzen, müssen Sie sich um die Erhaltung und Freigabe von Variablen nicht kümmern, Sie kön-

4 Sehen Sie sich den Anfang der Dokumentation an: Sie finden es unter *Conforms To* aufgeführt.

nen also `assign` überall dort verwenden, wo Sie entweder `assign` oder `retain` verwenden würden.[5]

Sie müssen ein weiteres Attribut setzen, wenn eine Eigenschaft als „nicht atomisch“ (`nonatomic`) deklariert werden soll. Standardmäßig sind Akzessoren atomisch. Bei einer atomischen Eigenschaft ist der Zugriff Thread-sicher. Bei einer Desktopanwendung, bei der Sie den Garbage Collector nutzen, müssen Sie dafür keinen hohen Preis zahlen. Wenn Sie nicht den Garbage Collector nutzen, müssen Sie im Getter einen Lock auf Objektebene einsetzen, den gewünschten Wert abrufen, den Lock freigeben und den abgerufenen Wert zurückliefern. Die Details für die Implemetierung eines atomischen Setters sind in dieser Situation ähnlich.

Wenn Sie Code für das iPhone entwickeln, steht Ihnen kein Garbage Collector zur Verfügung. Wenn Sie die Eigenschaften oft verwenden, werden Sie einen gewissen Performanceverlust feststellen. Beim Deklarieren von Eigenschaften für das iPhone neigen wir dazu, das Attribut `nonatomic` zu verwenden, während wir bei der Entwicklung für den Desktop die Standardeinstellung nutzen. Weitere Informationen finden Sie im Kapitel „Declared Properties“ in Apples *The Objective-C Programming Language* [App09f].

5.7 Übung: Eigenschaften hinzufügen

Fügen Sie `Greeter` die folgenden drei Eigenschaften hinzu: ein `NSInteger` namens `age`, einen `Greeter` namend `buddy` und ein `BOOL` namens `upperCase`.

Im Rahmen dieser Übung gehen wir davon aus, dass Sie keinen Garbage Collector nutzen und daher zwischen `assign` und `retain` unterscheiden müssen.

Verwenden Sie die richtigen Attribute, sodass es einen Getter, aber keinen Setter für `age` gibt und der Getter für `upperCase` den Namen `isUpperCase` hat.

5 Ich gehe davon aus, das Sie Mac OS X-Anwendungen für Leopard (und höher) entwickeln und daher den Garbage Collector nutzen. Für Desktopanwendungen sind `assign` und `copy` die möglichen Speicherattribute.

5.8 Lösung: Eigenschaften hinzufügen

Die eigentliche Arbeit erfolgt in der Header-Datei. Sie müssen die drei Instanzvariablen und dann die Eigenschaften wie folgt deklarieren:

Properties/HelloWorld15/Greeter.h

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

@interface Greeter : NSObject {
    NSString *name;
    NSInteger age;
    Greeter *buddy;
    BOOL upperCase;
}
-(NSString *) greeting;
-(id) initWithName:(NSString *)name;
@property(copy) NSString *name;
@property(assign, readonly) NSInteger age;
@property(retain) Greeter *buddy;
@property(assign, getter=isUpperCase) BOOL upperCase;

@end
```

Die `name`-Variable ist ein `NSString`, der `NSCopying` beherrscht, sodass wir das `copy`-Attribut verwenden.

Sowohl `age` als auch `upperCase` sind Primitive, weshalb wir `assign` verwenden. Da wir für `age` nur einen Getter erzeugen, verwenden wir außerdem `readonly`. Um die korrekte Namenskonvention für boolesche Werte einzuhalten, legen wir den Getter `mit isUpperCase` fest, indem wir `getter=isUpperCase` angeben.

Die `buddy`-Eigenschaft ist ein `Greeter`. Es handelt sich um einen Zeiger auf ein Objekt, das `NSCopying` nicht beherrscht, weshalb wir `retain` als Speicherattribut verwenden.

Wir nehmen die Namen der drei neuen Eigenschaften in die `@synthesize`-Zeile der Implementierungsdatei hinein und trennen sie durch Kommata. Beachten Sie, dass wir auch die `upperCase`-Variable in `initWithName:` initialisiert haben.

Properties/HelloWorld15/Greeter.m

```
#import "Greeter.h"

@implementation Greeter
@synthesize name, age, buddy, upperCase;
-(NSString *) greeting {
     return [[NSString alloc] initWithFormat:@"Hello, %@!", self.name];
}
-(id) initWithName:(NSString *) newName {
```

```
    if (self =[super init]){
        self.name = newName;
        self.upperCase = YES;
    }
    return self;
}
-(id) init {
      return [self initWithName:@"World"];
}
-(NSString *)description {
    return [[NSString alloc] initWithFormat:@"name: %@ \n created: %@",
            name, [NSDate date]];
}
@end
```

Wenn Sie wollen können Sie auch separate `@synthesize`-Direktiven für die einzelnen Eigenschaften verwenden:

```
@synthesize name;
@synthesize age;
@synthesize buddy;
@synthesize upperCase;
```

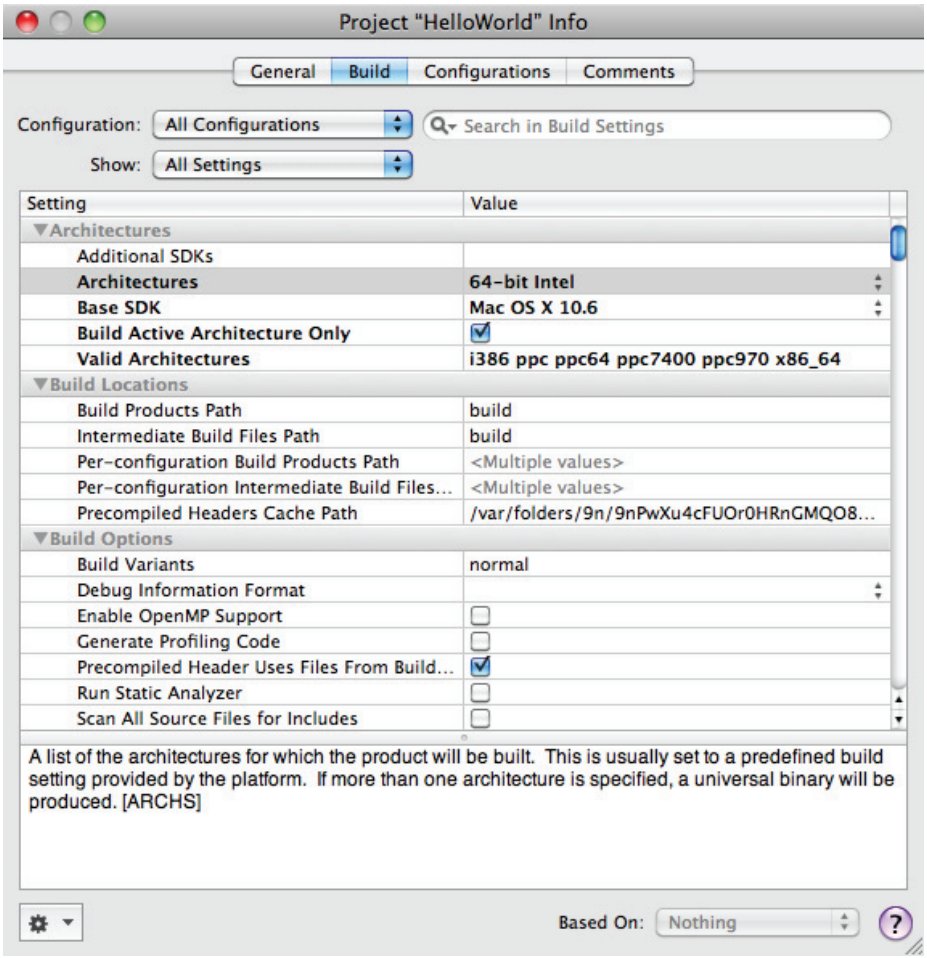
Eine letzte Vereinfachung unseres Codes steht noch aus.

5.9 Instanzvariablen entfernen

Wenn Sie auf einer 64-Bit-Maschine arbeiten oder nur für 64-Bit-Systeme entwickeln, können Sie die Instanzvariablen elminieren und diese vom Laufzeitsystem synthetisieren lassen. Leider wurde das für die 32-Bit-Laufzeit nicht portiert.

Da die 64-Bit-Laufzeit die zugrunde liegenen Instanzvariablen aus den `@property`- und `@synthesize`-Direktiven ableiten kann, werden Sie diese Informationen nicht einfügen, indem Sie die Instanzvariablen selbst deklarieren. Tatsächlich ist das sogar eine mögliche Fehlerquelle. Und wenn keine Instanzvariable hinter der Eigenschaft steht, kann der Compiler Ihnen Bescheid sagen, wenn Sie direkt auf die Variable zugreifen, statt über den Akzessor. Wenn Sie also versuchen, die Hintergrundfarbe mit `backgroundColor` statt mit `self.backgroundColor` zu setzen, kann der Compiler jetzt Ihren Fehler melden.

Lassen Sie uns also die Instanzvariablen entfernen, die hinter unseren Eigenschaften stehen. Zuerst überprüfen wir die Projekteinstellungen. Öffnen Sie sie in der Menüleiste über `Project` > `Edit Project Settings`. Wählen Sie den Build-Reiter aus, setzen Sie die *Architectures*-Einstellung auf *64-Bit Intel* und aktivieren Sie die Checkbox, um nur die aktive Architektur zu unterstützen.



Nun können Sie die Instanzvariablen aus der Header-Datei entfernen:

Properties/HelloWorld16/Greeter.h

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

@interface Greeter : NSObject {
}
-(NSString *) greeting;
-(id) initWithName:(NSString *)name;
@property(copy) NSString *name;
@property(assign, readonly) NSInteger age;
@property(retain) Greeter *buddy;
@property(assign, getter=isUpperCase) BOOL upperCase;

@end
```

Klicken Sie auf *Build & Run*, und die Anwendung sollte wie gewohnt laufen. Wenn Sie nach dem Entfernen der Instanzvariablen Warnmeldungen erhalten, überprüfen Sie, ob das Ganze unter 64 Bit läuft und Ihre Projekteinstellungen richtig sind.

Sie können den Eigenschaften zugrunde liegende Instanzvariablen entfernen, wenn Sie für das iPhone[6] entwickeln, aber nicht, wenn Ihre Programme unter der 32-Bit-Laufzeitumgebung von Mac OS X laufen sollen.

6 Während das hier geschrieben wurde, war es möglich, die Variablen auch bei der iPhone-Entwicklung entfernen, und der Code lief auf dem Gerät. Die Entfernung der Variablen wurde im Simulator aber nicht unterstützt.

Kapitel 6

Speicher

Sie haben gelernt, wie man Objekte im Interface Builder und mit Code erzeugt. Sie haben gesehen, wie man Nachrichten an Objekte sendet und Eigenschaften nutzt, um den Zustand einiger Objekte zu ändern. In diesem Kapitel lernen Sie die Regeln der Speicherverwaltung für Cocoa-Anwendungen kennen.

Wenn Sie ein Objekt erzeugen, nutzen Sie jedes Mal einen kleinen Speicherbereich. Wenn Sie dieses Objekt zu einem späteren Zeitpunkt nicht mehr benötigen, müssen Sie sicherstellen, dass dieser Speicherbereich nicht mehr verwendet wird. Über die Lebensdauer eines Programms hinweg können Sie Tausende oder Millionen von kurzlebigen Objekten erzeugen. Wenn Sie diese Objekte nicht aufräumen, verschwenden Sie Arbeitsspeicher (und, was noch schlimmer ist: machen ihn für andere Anwendungen unzugänglich). Als verantwortungsbewusste Entwickler stellen wir also sicher, dass diese überflüssigen Objekte wieder dem Pool freien Speichers zugeführt werden.

Wenn Sie nicht mehr genutzte Objekte nicht wieder freigeben, weist Ihre Anwendung ein Speicherleck (memory leak) auf. Mit der Zeit wird dieses Leck immer größer, manchmal sogar so groß, dass die Anwendung nicht mehr ausgeführt werden kann (oder die Ausführung anderer Programme behindert).

Wenn Sie andererseits ein Objekt freigeben, das immer noch verwendet wird, kann der Bereich durch anderen Programmcode überschrieben werden, der sich den Speicher greift – und das führt zu beschädigten Daten.

Und darum geht es in diesem Kapitel: Wie man das Ganze organisiert, damit der Speicher freigegeben wird, wenn man ihn nicht mehr benötigt, aber auch nicht vorher.[1]

6.1 Reference Counting

Reference Counting (also das Zählen von Referenzen) ist die Technik, die wir benutzen, um den Speicher mit den Objekten selbst zu verwalten. Die Regeln für das Speichermanagement sind leichter gesagt als getan:

- Wenn Ihnen ein Objekt gehört, sind Sie für seine Freigabe verantwortlich, wenn Sie es nicht mehr benötigen. Den Besitz eines Objekts zu beanspruchen, erhöht dessen Referenzzähler. Die Freigabe des Objekts reduziert den Zähler. Ein Objekt ist ungenutzt, wenn der Referenzzähler null ist.
- Wenn Ihnen ein Objekt nicht gehört, dürfen Sie es niemals freigeben.

Wenn Sie die erste Regel verletzen, verursachen Sie ein Speicherleck; verletzen Sie die zweite, können Sie ein Objekt freigeben, an das immer noch Nachrichten gesendet werden.

Stellt sich die Frage, wann einem ein Objekt gehört.

Jedes Mal, wenn Sie ein neues Objekt mit `alloc` erzeugen, gehört Ihnen das Objekt und Sie müssen es wieder freigeben, wenn Sie es nicht mehr verwenden. Es gibt auch eine Klassenmethode namens `new`, die eine Kombination aus `alloc` und `init` darstellt. Der Code

```
Greeter *host = [[Greeter alloc] init];
```

ist also identisch mit

```
Greeter *host = [Greeter new];
```

Wir neigen allerdings dazu, `new` bei Objective-C nicht zu verwenden, obwohl es zur Verfügung steht. Der springende Punkt dabei ist, dass Ihnen das Objekt gehört, wenn Sie es nutzen. Der Referenzzähler wird um 1 erhöht und Sie sind dafür verantwortlich, es wieder freizugeben, wenn Sie es nicht mehr benötigen.

---

1 Im Verlauf des Kapitels kann es passieren, dass Ihr Programm fehlschlägt, wenn es das nicht sollte, und nicht fehlschlägt, wenn es sollte. In diesem Fall wählen Sie `Build > Clean All Targets...` und führen es erneut aus.

Sie könnten auch Besitzer eines Objekts werden wollen, das an anderer Stelle erzeugt wurde. Wenn Sie an einem Objekt festhalten müssen, sind Sie dafür verantwortlich, ihm eine `retain`- oder `copy`-Nachricht zu senden. Das erhöht den Referenzzähler um 1. Egal, ob Sie ein Objekt mit `alloc` oder `new` erzeugen, oder ob Sie es mit `retain` oder `copy` festhalten – das Objekt gehört dann Ihnen und Sie sind für seine Freigabe verantwortlich, wenn Sie es nicht mehr benötigen.[2]

Wenn Sie das Objekt nicht mehr benötigen, sind Sie dafür verantwortlich, ihm eine `release`-Nachricht zu senden. Dadurch wird der Referenzzähler um 1 reduziert. Steht der Referenzzähler des Objekts auf null, wird sein `dealloc` aufgerufen, um die Ressourcen aufzuräumen und den Speicher freizugeben.

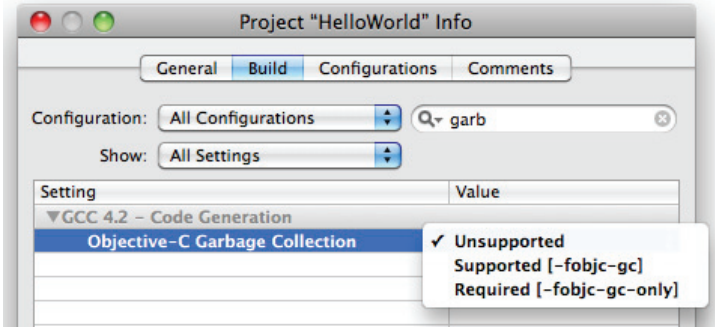
Sie rufen die `dealloc`-Methode nie direkt auf. Sie können ja nicht wissen, ob nicht noch jemand eine Referenz auf das Objekt hält, das Sie freigeben wollen. Sie rufen einfach `release` auf, um zu signalisieren, dass Sie an diesem Objekt nicht mehr interessiert sind. Wenn sich jeder an die Regeln hält, ist der Referenzzähler null, wenn sich keiner mehr für das Objekt interessiert, und `dealloc` wird aufgerufen.

Mehr gibt es dazu nicht zu sagen. Das sind die Regeln für die Erhöhung bzw. Reduzierung des Referenzzählers. Auch wenn die Regeln für die manuelle Speicherverwaltung einfach sind, finden Sie weitere Informationen in Apples *Memory Management Programming Guide for Cocoa* [App09e] und *Garbage Collection Programming Guide* [App08d].

6.2 Lecks mit dem Clang Static Analyzer aufspüren

Um die Sache bis zu diesem Punkt etwas zu vereinfachen, habe ich Sie gebeten, die automatische Garbage Collection einzuschalten. Um ein Speicherleck zu verursachen, müssen wir die Garbage Collection ausschalten. Wählen Sie `Project > Edit Project Settings` über das Menü und suchen Sie dann im Build-Reiter die Garbage Collector-Einstellung. Setzen Sie den Wert der Objective-C Garbage Collection auf *Unsupported*.

2 Sie werden gleich sehen, dass Sie ein Objekt nicht behalten, wenn Sie es mit einer Klassenmethode erzeugen. Es wird „Autoreleased“. Wenn Sie das neu erzeugte Objekt behalten wollen, müssen Sie das explizit über *retain* angeben.



Nun wollen wir ganz bewusst ein Speicherleck verursachen.

Wir beginnen damit, den GUI-Code aus `HelloWorldAppDel-egate` zu entfernen. Bereinigen Sie `HelloWorldAppDelegate.h` wie folgt:

Memory/HelloWorld17/HelloWorldAppDelegate.h

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

@interface HelloWorldAppDelegate : NSObject <NSApplicationDelegate> {
}
@end
```

Reduzieren Sie die Implementierungsdatei auf die Erzeugung eines `Greeter`-Objekts samt Logging:

Memory/HelloWorld17/HelloWorldAppDelegate.m

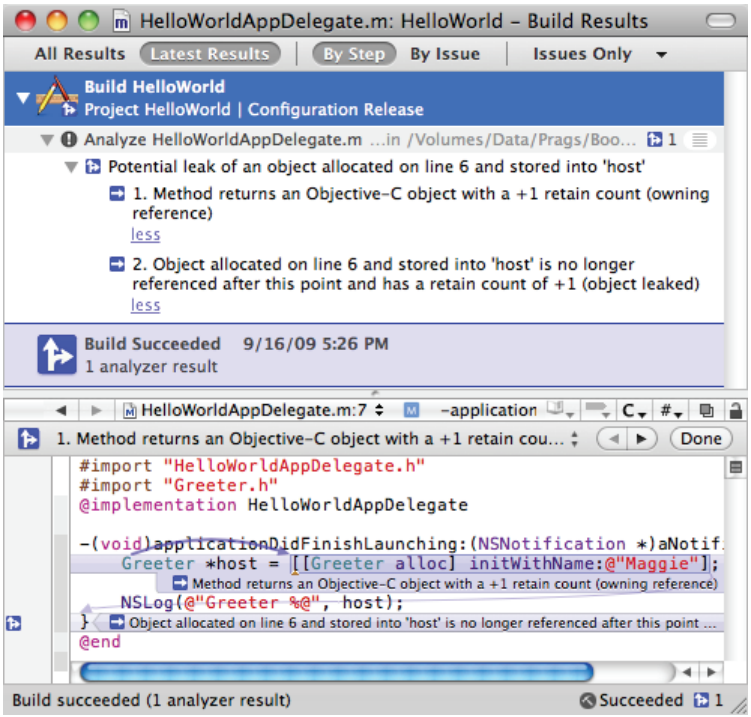
```
#import "HelloWorldAppDelegate.h"
#import "Greeter.h"

@implementation HelloWorldAppDelegate

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    Greeter *host = [[Greeter alloc] initWithName:@"Maggie"];
►   NSLog(@"Greeter %@", host);
}
@end
```

Wir sind für das Problem verantwortlich und wissen daher genau, was da schiefläuft. Wir haben eine Instanz der Klasse `Greeter` mit `alloc` erzeugt und dieses Objekt der Variablen `host` zugewiesen. Der Referenzzähler steht auf 1. Wir senden keine `release`-Nachricht an `host`, weshalb der Referenzzähler niemals null und der Speicher niemals freigegeben wird.

Das Programm ist sehr kurz und kurzlebig, weshalb es für uns kein großes Problem darstellt. Sie können sich vorstellen, dass es Teil eines wesentlich größeren Programms ist, und dass Sie nach der Fehler-

ursache suchen müssen. Bei Snow Leopard können Sie dieses Leck leicht finden, indem Sie den Clang Static Analyzer nutzen.

Sie starten den Clang Static Analyzer über die Menüleiste mit `Build` > `Build & Analyze` oder über das Tastaturkürzel ⇧⌘A.



Der Bericht zeigt uns, dass das Objekt alloziert und in `host` gespeichert wurde, und das sein retain-Zähler auf 1 steht. Anstelle des im obigen Bild dargestellten einzelnen Fehlers könnten auch drei Probleme aufgeführt werden. Wenn Sie jedes von ihnen anklicken, werden Sie erkennen, dass sie alle denselben Fehler ansprechen – und wenn Sie ihn korrigieren, verschwinden auch alle. Nachdem `host` in der Loganweisung benutzt worden ist, wird die Variable nicht mehr referenziert. Das ist unser Speicherleck. Das `Greeter`-Objekt wird instanziiert und verwendet und hängt untätig rum, solange das GUI-Fenster geöffnet bleibt.

6.3 Das Speicherleck unter Mac OS X beheben

Um dieses Speicherleck unter Mac OS X zu beheben, kehren Sie in die Projekteinstellungen zurück und ändern die Einstellung für die Objective-C Garbage Collection von *Unsupported* in *Required*.

Klicken Sie auf *Build & Analyze*, und das Speicherleck ist verschwunden.

Wow, Sie haben gerade eine neue Möglichkeit der Speicherverwaltung kennengelernt. Keine Referenzzählung und kein Austarieren von `new`, `alloc`, `retain` oder `copy` durch `release`. In diesem Buch setze ich voraus, dass Sie Leopard (oder höher) einsetzen, und daher ist der Rat ganz einfach:

Schalten Sie die Garbage Collection ein.

Diejenigen, die das Zählen von Referenzen gemeistert haben, betrachten den Garbage Collector als Affront gegen Programmierer. Das ist aber nicht der Fall. Wenn Sie für Mac OS X entwickeln, schalten Sie den Garbage Collector ein und lassen Sie das Reference Counting hinter sich.

Das erste System mit Garbage Collector, mit dem ich gearbeitet habe, war Java. In den ganz frühen Java-Tagen kam alles zum Halten, wenn der Garbage Collector lief. Tatsächlich erschien eine Animation auf dem Bildschirm. Sie zeigte einen Bulldozer, der irgendwelchen Kram zusammenschiebt.

Eine Zeit lang haben die Leute den vom Compiler erzeugten Bytecode untersucht und nach kleinen Optimierungen Ausschau gehalten. Eine dieser Optimierungen bestand darin, Schleifen so umzukehren, dass sie nicht von `0` bis zu einem bestimmten Limit liefen: Es war effektiver, wenn Sie von diesem Limit auf `0` dekrementiert wurden.

Doch während wir diese kleinen Hacks herausbekamen, wurde der Garbage Collector für natürlich entstehende Situationen immer besser. Es stellte sich heraus, dass unsere cleveren Hacks dem Garbage Collector das Leben schwer machten.

Auf dieser Ebene bewegen wir uns nun in etwa mit Cocoa auf dem Desktop.

Doch unabhängig davon müssen Sie wissen, wie die Referenzzählung funktioniert. Wir werden uns gleich ein iPhone-Beispiel ansehen, bei dem Ihnen gar nichts anderes übrig bleibt, als den Speicher manuell zu verwalten. Zuerst möchte ich Ihnen zeigen, was der Garbage Collector für Sie tun bzw. nicht tun kann, wenn es um die Speicherattribute von Eigenschaften geht.

6.4 Eigenschaften und Garbage Collection

Nachdem wir nun die Garbage Collection eingeschaltet haben, wollen wir uns die Deklarationen der Eigenschaften in der `Greeter`-Header-Datei noch einmal ansehen:

Memory/HelloWorld17/Greeter.h

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

@interface Greeter : NSObject {
}

-(NSString *) greeting;
-(id) initWithName:(NSString *)name;

@property(copy) NSString *name;
@property(assign, readonly) NSInteger age;
@property(retain) Greeter *buddy;
@property(assign, getter=isUpperCase) BOOL upperCase;

@end
```

Wenn jemand das Speichermanagement für uns übernimmt, können wir dann nicht einfach die Speicherattribute aus den Deklarationen der Eigenschaften entfernen?

Memory/HelloWorld18/Greeter.h

```
@property NSString *name; //this line is not correct
@property(readonly) NSInteger age;
@property Greeter *buddy;
@property(getter=isUpperCase) BOOL upperCase;
```

Wenn Sie den Compiler laufen lassen, werden Sie sehen, dass die Antwort lautet: „Größtenteils schon."

Erinnern Sie sich daran, dass das Standard-Speicherattribut `assign` lautet, sodass Sie die Attribute für `age` und `upperCase` bereits hätten weglassen können. Ist die Garbage Collection aktiv, gibt es effektiv keinen Unterschied zwischen `retain` und `assign`, weshalb Sie das Speicherattribut für `buddy` nicht mehr angeben müssen.

Bei der Kompilierung erhalten Sie die folgende Warnung:

```
Default 'assign' attribute on property 'name' which implements
'NSCopying' not appropriate with -fobjc-gc-only.
```

Das Problem besteht darin, dass `name` zwar als `NSString` deklariert ist, aber auch vom Typ `NSMutableString` sein könnte, der `NSString` erweitert. Ein `NSString` ist unveränderlich, Sie können also `copy` verwenden, aber nicht `retain`. Für `NSMutableString` würden Sie dagegen `retain`

verwenden, obwohl beide funktionieren würden. Aufgrund dieser möglichen Mehrdeutigkeit müssen Sie das Speicherattribut für `name` explizit deklarieren.

Memory/HelloWorld19/Greeter.h

```
► @property(copy) NSString *name;
@property(readonly) NSInteger age;
@property Greeter *buddy;
@property(getter=isUpperCase) BOOL upperCase;
```

Der Compiler wird Sie warnen, wenn es zu solchen Problemen kommt. Meistens werden Sie sich dabei ertappen, `copy` explizit zu deklarieren, wenn Sie mit `NSString`, aber auch mit Collection-Klassen wie `NSArray`, `NSDictionary` und `NSSet` arbeiten. Alle diese Klassen besitzen als Subklasse eine veränderliche (mutable) Variante.

Machen Sie sich keine Sorgen, wenn es noch nicht Klick gemacht hat. Ich wollte hauptsächlich, dass Sie darauf vorbereitet sind, wenn der Compiler sich später in diesem Buch über dieses Problem beschwert. Gedanken um die Speicherverwaltung müssen Sie sich hauptsächlich machen, wenn Sie Anwendungen für iPhone, iPod touch oder iPad entwickeln. Der Rest dieses Kapitels widmet sich einer iPhone-App.

6.5 Eine Taschenlampe entwickeln

Wenn Sie für iPhone oder iPad entwickeln, müssen Sie Reference Counting verwenden und den Speicher selbst verwalten. Lassen Sie uns die iPhone-Variante unseres *HelloWorld*-Projekts entwickeln, damit wir mit einigen der gezeigten Regeln experimentieren können.

Sie müssen sich auf *http://developer.apple.com/iphone* als iPhone-Entwickler registrieren lassen. Die Registrierung ist kostenlos, aber Sie müssen Apples Nutzungsbedingungen zustimmen. Wenn Sie so weit sind, Anwendungen auf dem iPhone zu nutzen oder diese im App-Store zu vertreiben, müssen Sie einem der kostenpflichtigen Programme beitreten.

Wenn Sie nicht als iPhone-Entwickler registriert sind, können Sie die Arbeit einfach mit dem aktuellen Beispiel fortsetzen. Sie müssen die Garbage Collection nur wieder auf *Unsupported* zurückstellen.

Legen Sie ein neues Projekt in Xcode an (`File` > `New Project...`). Diesmal wählen Sie die Vorlage `iPhone` > `Application` > `Window-based Application`. Stellen Sie sicher, dass die Checkbox für die Verwendung von Core Data deaktiviert ist, und klicken Sie auf *Choose*. Nennen Sie das Projekt *Flashlight* und speichern Sie es mit *Save*.

Klicken Sie auf *Build & Run*, und der iPhone-Simulator wird gestartet und die Anwendung ausgeführt. Es erscheint ein weißer Bildschirm mit einer Statuszeile. Herzlichen Glückwunsch, Sie haben eine Taschenlampe entwickelt. Einige Leute haben so etwas sogar im App-Store verkauft.

Wir müssen die Header- und Implementierungsdateien der `Greeter`-Klasse kopieren. Wählen Sie den Ordner `Classes` in *Groups & Files*. Wählen Sie den Menüpunkt `Project > Add to Project...` Bewegen Sie sich an den Ort, an dem Sie das `HelloWorld`-Projekt abgelegt haben, wählen Sie `Greeter.h` und `Greeter.m` aus und klicken Sie den *Add*-Button an. Klicken Sie bei Bedarf die Checkbox zum Kopieren der Elemente in den Flashlight-Ordner an, und klicken Sie auf den *Add*-Button. Sie haben die Klasse `Greeter` zu diesem Projekt hinzugefügt.

Ihre `FlashlightAppDelegate.m` sollte im Wesentlichen mit `HelloWorldAppDelegate.m` übereinstimmen:

Memory/Flashlight1/Classes/FlashlightAppDelegate.m

```
#import "FlashlightAppDelegate.h"
#import "Greeter.h"

@implementation FlashlightAppDelegate
@synthesize window;

-(void)applicationDidFinishLaunching:(UIApplication *)application {
    Greeter *host = [[Greeter alloc] initWithName:@"Maggie"];
    NSLog(@"Greeter %@", host);
}
@end
```

Klicken Sie auf *Build & Run*: Sie werden sehen, dass es einige Probleme zu lösen gilt.

Zuerst erhalten wir einen Fehler in `Greeter.h`, dass es keine Datei bzw. keinen Ordner namens `Cocoa/Cocoa.h` gibt. Für das iPhone heißt das Framework, das die GUI einbindet, `UIKit/UIKit.h`. Da hier keine grafischen Elemente im Spiel sind, importieren wir nur das Foundation-Framework. Ersetzen Sie

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

in `Greeter.h` durch

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

Das andere Problem ist ein Artefakt des Simulators. Wir müssen unsere Instanzvariablen explizit deklarieren. Hier die korrigierte `Greeter.h`:

Memory/Flashlight1/Classes/Greeter.h

```
#import <Foundation/Foundation.h>
@interface Greeter :NSObject {
►    NSString *name;
►    NSInteger age;
►    NSDate *today;
►    BOOL upperCase;
}
-(NSString *) greeting;
-(id) initWithName:(NSString *)name;

@property(copy) NSString *name;
@property(assign, readonly) NSInteger age;
► @property(copy) NSDate *today;
@property(assign, getter=isUpperCase) BOOL upperCase;

@end
```

Ich habe auch das `buddy`-Objekt vom Typ `Greeter` durch ein `NSDate`-Objekt namens `today` ersetzt.[3] Ich werde es verwenden, um einen Timestamp zu generieren, wenn das `Greeter`-Objekt erzeugt wird.

Memory/Flashlight1/Classes/Greeter.m

```
#import "Greeter.h"

@implementation Greeter

► @synthesize name, age, today, upperCase;
-(NSString *) greeting {
    return [[NSString alloc] initWithFormat:@"Hello, %@!", self.name];
}
-(id) initWithName:(NSString *) newName {
   if (self =[super init]){
       self.name = newName;
       self.upperCase = YES;
►      self.today = [NSDate date];
   }
   return self;
}
-(id) init {
   return [self initWithName:@"World"];
}
-(NSString *)description {
    return [[NSString alloc] initWithFormat:@"name: %@ \n created: %@",
►            self.name,self.today];
}
@end
```

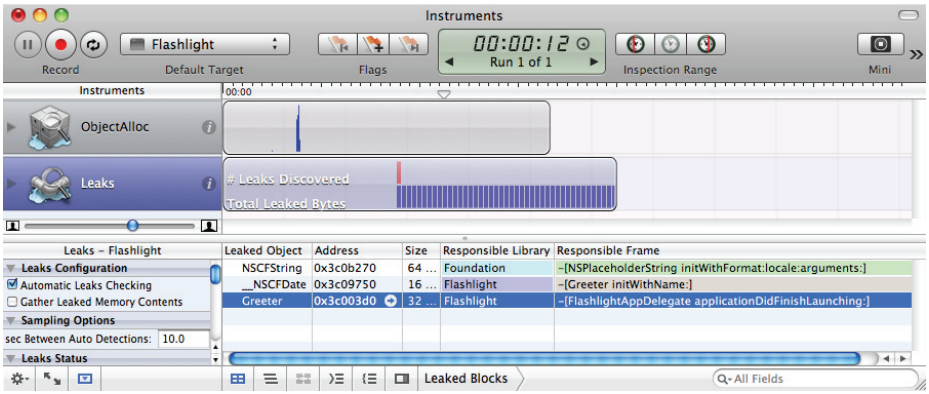
3 Ich habe den convenience constructor `date` anstelle von `alloc` und `init` verwendet. Am Ende des Kapitels werden Sie verstehen warum.

Klicken Sie auf *Build & Run*, und Sie sollten die „Taschenlampe“ mit der gleichen Ausgabe wie in der Desktopversion sehen. Klicken Sie im Simulator auf *Build & Analyze*, und Sie sollten die gleichen Speicherlecks sehen wie vorhin.[4] Wir konzentrieren uns auf das in `FlashlightAppDelegate.m`.

6.6 Lecks mit Instruments aufspüren

Man könnte (und sollte) ganze Bücher über Apples Performance- und Debugging-Tools schreiben. Ich möchte Ihnen kurz zeigen, wie man Instruments hätte nutzen können, um das Speicherleck in `FlashlightAppDelegate.m` aufzuspüren.

Sie haben *Build & Run* bereits ausgeführt, es gibt also ein Programm, das wir unter Instruments ausführen können. In Xcode wählen Sie den Menüpunkt `Run > Run with Performance Tool > Leaks`. Instruments wird gestartet, und die Flashlight-Anwendung wird innerhalb von Instruments ausgeführt. Fast sofort sollten Sie eine vertikale blaue Linie in der oberen Zeile des Instruments-Fensters sehen, während die Objekte alloziert werden. Nach etwa zehn Sekunden sehen Sie, während die Objekte alloziert werden, eine rote Zeile mit einer Reihe blauer Zeilen in der zweiten Reihe des Instruments-Fensters. Die rote Zeile repräsentiert die Anzahl der entdeckten Lecks, und die blauen Zeilen die Bytes, die verloren gegangen sind.



4 Ich gehe davon aus, dass Sie alle Performance-Tools im Simulator laufen lassen. Sie stehen Ihnen kostenlos zur Verfügung. Sie müssen aber zahlen, wenn Sie Apps auf Ihrem Gerät nutzen wollen.

Das Leck tritt wesentlich früher ein, aber laut Voreinstellung wird nur alle zehn Sekunden nach Lecks gesucht. Standardmäßig werden die Allozierungsdaten des Objekts in der unteren rechten Ecke des Instruments-Fensters ausgegeben. Wählen Sie oben links das Leaks-Instrument aus, und Sie sehen die Informationen zum Speicherleck. Sie erkennen in der Ausgabe, dass der Ursprung des Lecks in der `FlashlightAppDelegate`-Methode `applicationDidFinishLaunching:` liegt. Diese Information reicht aus, um das Leck zu finden. Wenn Sie die Zeile doppelt anklicken, die das leckende `Greeter`-Objekt anzeigt, gelangen Sie an die entsprechende Stelle im Quellcode. Da wir das Leck nun kennen, wollen wir es beheben.

6.7 Das Speicherleck auf dem iPhone beheben

Das Speichermanagement ist ein Kompromiss zwischen vielen Faktoren. Das Wichtigste ist, dass Sie den Speicher für ein Objekt nicht freigeben sollten, während es jemand noch nutzt. Andererseits wollen Sie kein Speicherleck heraufbeschwören, indem Sie an einem Objekt festhalten, das niemand mehr benötigt. Dieses Ziel wird bei Objective-C mit Reference Counting erreicht, dem Zählen von Referenzen.

Wir haben kurz angesprochen, dass der Referenzzähler auf 1 gesetzt wird, wenn Sie ein `Greeter`-Objekt über `alloc` erzeugen:

```
Greeter *host = [[Greeter alloc] initWithName:@"Maggie"];
```

Das ist genau der Fehler, den uns *Build & Analyze* meldet. Es weiß, das Sie den Referenzzähler auf 1 gesetzt, die Variable dann aber nie verwendet haben. Die Freigabe geschieht folgendermaßen:

```
[host release];
```

Damit wird der Referenzzähler um 1 dekrementiert. In unserem Fall reicht das aus. Der Referenzzähler für `host` wird null, und die `dealloc`-Methode wird aufgerufen, um die Ressourcen des Objekts aufzuräumen und den Speicher freizugeben. Sie senden die `release`-Nachricht an das `host`-Objekt, sobald klar ist, dass Sie es nicht mehr benötigen. In unserem Beispiel würde das so aussehen:

Memory/Flashlight2/Classes/FlashlightAppDelegate.m

```
-(void)applicationDidFinishLaunching:(UIApplication *)application {
    Greeter *host = [[Greeter alloc] initWithName:@"Maggie"];
    NSLog(@"Greeter %@", host);
►   [host release];
}
```

Klicken Sie auf *Build & Analyze*, und das Speicherleck ist verschwunden. Als Nächstes nehmen wir uns etwas Zeit, um uns den zweiten typischen Fehler im Zusammenhang mit der Speicherverwaltung anzusehen: Was passiert, wenn Sie versuchen, eine Nachricht an ein bereits freigegebenes Objekt zu senden?

6.8 Zombies nutzen

Bisher haben wir ein Objekt vom Typ `Greeter` erzeugt, seinen Inhalt in der Konsole ausgegeben und das Objekt dann wieder freigegeben. Jetzt wollen wir dem Objekt ganz bewusst noch eine weitere Nachricht senden. Wir geben seinen Inhalt einfach noch einmal in der Konsole aus:

Memory/Flashlight3/Classes/FlashlightAppDelegate.m

```
-(void)applicationDidFinishLaunching:(UIApplication *)application {
    Greeter *host = [[Greeter alloc] initWithName:@"Maggie"];
    NSLog(@"Greeter %@", host);
    [host release];
►   NSLog(@"Greeter %@", host);
}
```

Wählen Sie in der Menüleiste `Build > Build & Debug`. Sie sollten eine Standardmeldung im Konsolenfenster sehen, gefolgt von so etwas wie dem hier:[5]

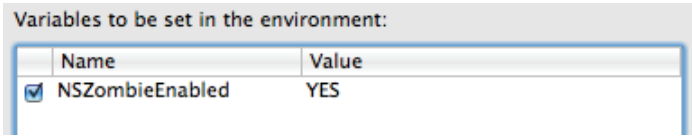
```
Greeter name: Maggie
created: 2009-09-18 14:24:43 -0400
objc[11179]: FREED(id): message respondsToSelector:
                        sent to freed object=0x3b06760
Program received signal: ''EXC_BAD_INSTRUCTION''.
```

Das hilft uns gar nicht weiter. Wir wissen, welches Objekt freigegeben wurde, weil wir es zu Demonstrationszwecken selbst freigegeben haben, aber bei einem großen Programm, wo die Objekte kommen und gehen, wäre es viel schwieriger, das richtige Objekt zu finden.

Wir werden Zombies nutzen, um uns hier aus der Patsche zu helfen. Um diese zu aktivieren, wählen Sie `Executables > Flashlight` in *Groups & Files*. Klicken sie die Datei mit der rechten Maustaste an und wählen Sie *Get Info* oder drücken Sie (bei aktiviertem `Flashlight`) das Tastaturkürzel ⌘ I. Wählen Sie dann den *Arguments*-Reiter, und am unteren Rand sehen Sie den Text „Variables to be set in the environment:“.

5 Wenn Sie diese Nachricht nicht sehen, versuchen Sie es mit `Build > Clean All Targets` und klicken Sie erneut auf *Build & Run*.

Fügen Sie eine Umgebungsvariable hinzu, indem Sie den +-Button in der unteren linken Ecke anklicken. Tragen Sie *NSZombieEnabled* als Namen der Variable ein und setzen Sie seinen Wert auf *YES*.



Schließen Sie das Fenster und führen Sie *Build & Debug* erneut aus. Diesmal sollten Sie in der Konsole Folgendes sehen:

```
Greeter name: Maggie
 created: 2009-09-18 16:13:57 -0400
*** -[Greeter respondsToSelector:]:
               message sent to deallocated instance 0x380d950
```

Diesmal hilft uns der Zombie dabei, zu erkennen, dass der Objekttyp ein `Greeter` ist. Diese Technik ist sehr hilfreich, wenn Sie beim Debuggen herauszufinden versuchen, welchem freigegebenen Objekt Sie Nachrichten senden. Bevor Sie weitermachen, müssen Sie die Variable wieder löschen, die `NSZombie` aktiviert. Sie werden die Objekte nicht weiter festhalten wollen, wenn Sie die Anwendung aktiv einsetzen.

6.9 Aufräumen in dealloc

Sobald der Referenzzähler für ein Objekt den Wert null erreicht, wird sein `dealloc` aufgerufen. Hier räumen Sie die von Ihrem Objekt genutzten Ressourcen auf. Zum Beispiel gibt es in `Greeter` zwei Zeiger auf Objekte, die uns gehören: `name` und `today`. Wir müssen jedem ein *release* senden, wenn unser `Greeter`-Objekt freigegeben werden soll. Dazu überschreiben wir die `dealloc`-Methode:

Memory/Flashlight5/Classes/Greeter.m

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

Beachten Sie, dass wir in der `init`-Methode zuerst die Superklasse ihre Initialisierung durchführen lassen, bevor wir unsere eigene Initialisierung vornehmen. Hier räumen wir zunächst unsere eigenen Objekte auf und rufen erst dann `[super dealloc]` auf.

Sie können die folgende Zeile in Ihr `dealloc` einfügen, um eine kurze Meldung in der Konsole auszugeben, wenn ihr `Greeter`-Objekt freigegeben wird.

```
NSLog(@"In Greeter dealloc.");
```

Wenn Sie ein ganz Genauer sind und sichergehen wollen, dass es Ihnen auf keinen Fall passieren kann, dass Sie eine Nachricht an ein freigegebenes Objekt senden, dann können Sie die Variablen auf `nil` setzen, nachdem Sie sie freigegeben haben:

Memory/Flashlight6/Classes/Greeter.m

```
-(void) dealloc {
    [name release];
    name = nil;
    [today release];
    today = nil;
    [super dealloc];
}
```

Hier geben wir eine Instanzvariable frei und setzen sie auf `nil`. Sie können diese Schritte kombinieren, indem Sie die entsprechenden Eigenschaften benutzen:

Memory/Flashlight7/Classes/Greeter.m

```
-(void) dealloc {
    self.name = nil;
    self.today = nil;
    [super dealloc];
}
```

Wie Sie im nächsten Abschnitt sehen werden, gibt `self.name = nil` den `name`n frei und setzt ihn auf `nil`.[6]

6.10 Retain und Release in einem Setter

Es folgt ein Beispiel für das *retain/release*-Muster, das Sie beim Ändern eines Wertes in einem Objekt verwenden werden.[7] Nehmen wir an, Sie haben bereits ein Objekt namens `greeter` vom Typ `Greeter` erzeugt. Eine idiomatische `setGreeter:`-Methode mit *retain/release*-Muster würde so aussehen:

6 Das hat möglicherweise Konsequenzen, wenn Sie das Key Value Observing nutzen und jemand anders noch Änderungen an diesem freigegebenen und auf `nil` gesetzten Objekt beobachtet. Ich neige dazu, der Variablen nur ein `release` im `dealloc` zu senden und es dabei zu belassen.

7 Wir haben das kurz gesehen, als wir uns im vorigen Kapitel mit den Speicherattributen für Eigenschaften befasst haben. Diese Version ist ein bisschen anders, und nun können Sie den *retain/release*-Zyklus besser verstehen.

```
-(void) setGreeter: (Greeter *) newGreeter {
   if (newGreeter != greeter) {
      [newGreeter retain];
      [greeter release];
      greeter = newGreeter;
   }
}
```

Ihnen wurde das Objekt `newGreeter` gesendet. Es soll Ihnen gehören, damit Sie etwas damit anstellen können, weshalb Sie es über *retain* festhalten müssen. Andererseits sind Sie im Begriff, die Variable `greeter` auf das durch `newGreeter` referenzierte Objekt zeigen zu lassen.

Zuerst stellen Sie sicher, dass die Zeiger nicht auf dieselbe Speicherstelle verweisen. Ist das der Fall, gibt es nichts zu tun. Wenn nicht, lassen Sie `greeter` auf das durch `newGreeter` referenzierte Objekt verweisen.

Sie benötigen das Objekt, auf das `newGreeter` verweist, also senden Sie eine `retain`-Nachricht. Gleichzeitig benötigen Sie das Objekt nicht mehr, auf das `greeter` zeigt, und geben es per *release* frei. Nun lassen Sie `greeter` auf das Objekt verweisen, auf das `newGreeter` zeigt. Zwar werden Sie die Variable `newGreeter` nicht mehr benutzen, aber Sie werden das verwenden, worauf sie zeigt. Bei diesem ganzen *retain*/*release*-Zyklus geht es um das Festhalten und Freigeben dessen, worauf die Variablen zeigen, und nicht um die Variablen selbst.

Diese *retain*/*release*-Methode des Speichermanagements weist Parallelen zum richtigen Leben auf. Zum Beispiel war einmal das Hotel komplett ausgebucht, in dem ich unbedingt übernachten wollte. Man setzte mich auf eine Warteliste, und ich nahm mir ein Zimmer in einem nahegelegenen Hotel. Alle Informationen trug ich in iCal ein. Glücklicherweise wurde in meinem Wunschhotel ein Zimmer frei. Denken Sie einen Augenblick über die Reihenfolge nach, in der man die Änderungen vornimmt, und Sie werden erkennen, dass sie genau dem entspricht, wie die Speicherverwaltung in Objective-C abläuft.

Zuerst habe ich eine Reservierung für das neue Zimmer in meinem Wunschhotel erhalten. Dann habe ich die existierende Reservierung im Ausweichhotel freigegeben. Zum Schluss habe ich die Daten zur geänderten Reservierung in iCal aktualisiert. Wenn wir uns das iCal-Ereignis als unsere Variable vorstellen, habe ich das Objekt festgehalten (retain), auf das die Variable zeigen soll, das Objekt freigegeben, auf das sie momentan zeigt (release), und habe die Variable dann auf das neue Objekt zeigen lassen.[8]

8 Dehnt man diese Metapher für unser Reservierungsbeispiel noch ein wenig aus, entspricht das Einschalten eines Reisebüros der automatischen Garbage Collection.

Wenn Sie für `greeter` eine Eigenschaft mit dem Speicherattribut `retain` verwenden, folgt der während der Kompilierung erzeugte Setter diesem Muster. Das ist ein weiterer Grund dafür, dass Sie die Vorteile von Eigenschaften nutzen sollten, statt direkt auf die darunterliegenden Instanzvariablen zuzugreifen: Lassen Sie sich von den Eigenschaften bei der korrekten Speicherverwaltung unterstützen.

6.11 Der Autorelease-Pool

Es gibt noch eine weitere Situation, um die wir uns bisher nicht gekümmert haben. Um das Problem zu zeigen, erzeugen wir ein Instanz des `Greeter`-Objekts in einer separaten Methode in `FlashlightAppDelegate.m`.

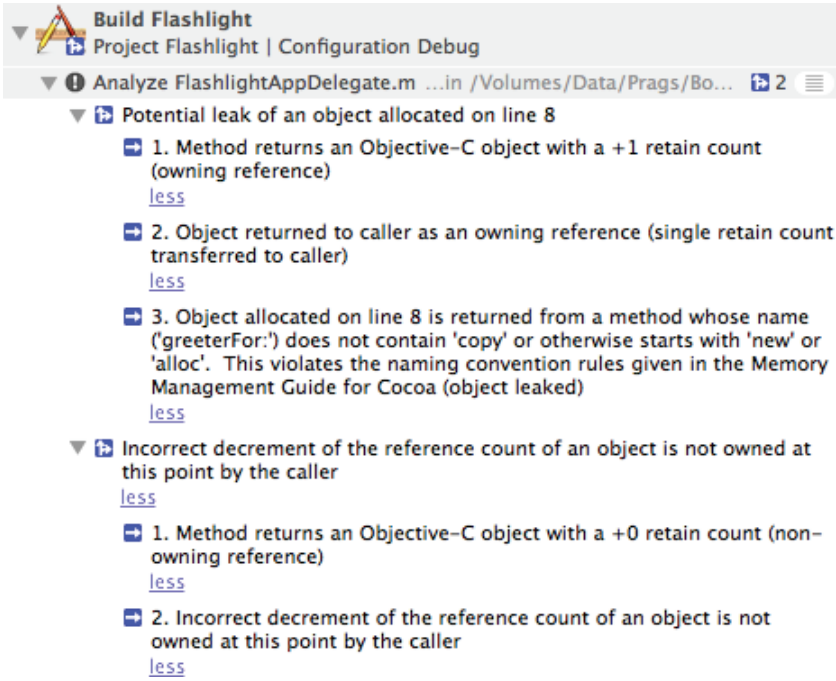
Memory/Flashlight8/Classes/FlashlightAppDelegate.m

```
#import "FlashlightAppDelegate.h"
#import "Greeter.h"

@implementation FlashlightAppDelegate
@synthesize window;

-(Greeter *) greeterFor:(NSString *) personName {
    return [[Greeter alloc] initWithName:personName];
}

-(void)applicationDidFinishLaunching:(UIApplication *)application {
    Greeter *host = [self greeterFor:@"Maggie"];
    NSLog(@"Greeter %@", host);
    [host release];
}
@end
```

Klicken Sie auf *Build & Run*, und die Anwendung verhält sich korrekt. Sie erzeugt ein neues Objekt, gibt es korrekt aus, gibt es dann frei und ruft `dealloc` auf.

```
Greeter name: Maggie
 created: 2009-09-17 14:12:09 -0400
In Greeter dealloc.
```

Starten Sie statt desssen *Build & Analyze*, werden zwei Probleme in `FlashlightAppDelegate.m` angezeigt. Dieses Problem begegnet Ihnen üblicherweise, wenn die Methode, die ein Objekt zurückliefert (in unserem Fall `greeterFor:( )`) in einer anderen Klasse definiert ist als das Objekt, das es aufruft. Bei diesem Beispiel sind beide Hälften der Transaktion im selben Objekt enthalten, weshalb es erfolgreich erzeugt und wieder freigegeben werden konnte. Der Clang Static Analyzer erkennt ein potenzielles Problem, indem er sich die beiden Methoden `greeterFor:( )` und `application-DidFinishLaunching:( )` getrennt ansieht.



Der zweite Fehler sagt uns, dass wir ein Objekt freigegeben haben, das uns nicht gehört. Wenn Sie sich die `applicationDidFinishLaunching:`-Methode ansehen, können Sie tatsächlich erkennen, dass wir `host` freigegeben haben, ohne wissen zu können, ob es uns gehört.

Wir wollen dieses Problem für den Moment ignorieren und uns auf den anderen Punkt konzentrieren. Wenn Sie sich den Code ansehen, erkennen Sie, dass uns `host` gehört, weil wir es mit `alloc`: angelegt haben.

Memory/Flashlight8/Classes/FlashlightAppDelegate.m

```
-(Greeter *) greeterFor:(NSString *) personName {
    return [[Greeter alloc] initWithName:personName];
}
```

Um das Problem zu verdeutlichen, erzeugen wir in zwei separaten Schritten das Objekt und geben es dann frei:

Memory/Flashlight9/Classes/FlashlightAppDelegate.m

```
-(Greeter *) greeterFor:(NSString *) personName {
    Greeter *tempGreeter = [[Greeter alloc] initWithName:personName];
    return tempGreeter;
}
```

Wo können wir `tempGreeter` freigeben?

Sie können `tempGreeter` nicht freigeben, bevor Sie es zurückgeliefert haben, da der Zähler dann null und `tempGreeter` derefenziert wird, bevor Sie es zurückgeben können. Sie können die Referenz für `tempGreeter` aber nicht ignorieren. Das ist das Speicherleck, das wir zu beheben versuchen. Wenn Sie das Objekt nicht freigeben, bleibt `tempGreeter` erhalten, obwohl das Objekt nicht mehr verwendet wird, weil Sie den *retain*-Zähler nicht dekrementiert haben.[9]

Die Lösung liegt in der Verwendung eines Autorelease-Pools.[10] Sie senden `tempGreeter` die `autorelease`-Nachricht, und es wird als freigegeben gekennzeichnet, sobald die die `greeterFor:`-Methode aufrufende Methode abgeschlossen wird. Sie können das erledigen, wenn Sie die `Greeter`-Instanz erzeugen. In diesem Fall können Sie auf die temporäre Variable verzichten und Ihr Ziel wie folgt erreichen:

Memory/Flashlight10/Classes/FlashlightAppDelegate.m

```
-(Greeter *) greeterFor:(NSString *) personName {
    return [[[Greeter alloc] initWithName:personName] autorelease];
}
```

Nun können wir unser zweites Problem lösen, indem wir den Aufruf von `[host release]` in `applicationDidFinishLaunching:` entfernen. Das Objekt, auf das die `host`-Variable zeigt, bleibt jetzt nicht erhalten, weil es nun per *autorelease* automatisch freigegeben wird. Da uns das Objekt nicht mehr gehört, dürfen wir es auch nicht mehr freigeben.

6.12 Bequemlichkeitskonstruktoren verwenden

In `Greeter.m` lauern in den `greeting`- und `description`-Methoden immer noch Speicherlecks. Sehen wir uns die `greeting`-Methode an:

Memory/Flashlight10/Classes/Greeter.m

```
-(NSString *) greeting {
    return [[NSString alloc] initWithFormat:@"Hello, %@!", self.name];
}
```

Das Problem ist das gleiche wie das, das wir gerade gelöst haben. Wir könnten das Speicherleck beheben, indem wir dem zurückgegebenen Objekt ein `autorelease` senden. Wie sich zeigt, bietet Apple aber eine andere Lösung.

9 Es ist wohl offensichtlich, dass die Freigabe des Objekts nicht *nach* dem `return` erfolgen kann.

10 Weitere Informationen zu Autorelease-Pools und um Speichermanagement finden Sie in Apples *Memory Management Programming Guide for Cocoa*.

Hier sehen Sie eine Liste mit Methoden, die `NSString` zur Erzeugung und Initialisierung von Strings zur Verfügung stellt:

Creating and Initializing Strings

```
+ string
- init
- initWithBytes:length:encoding:
- initWithBytesNoCopy:length:encoding:freeWhenDone:
- initWithCharacters:length:
- initWithCharactersNoCopy:length:freeWhenDone:
- initWithString:
- initWithCString:encoding:
- initWithUTF8String:
- initWithFormat:
- initWithFormat:arguments:
- initWithFormat:locale:
- initWithFormat:locale:arguments:
- initWithData:encoding:
+ stringWithFormat:
+ localizedStringWithFormat:
+ stringWithCharacters:length:
+ stringWithString:
+ stringWithCString:encoding:
+ stringWithUTF8String:
```

Die Methoden, denen ein + voransteht, sind Klassenmethoden, und die mit dem - sind Instanzmethoden und müssen zusammen mit `alloc` eingesetzt werden. Für Klassenmethoden gibt es immer auch als Instanzmethoden ausgelegte Versionen. Zum Beispiel ist `stringWithString:` mit `initWithString:` gepaart und `stringWithUTF8String:` mit `initWithUTF8String:`.

Statt also mit

```
[[NSString alloc] initWithFormat:@"Hello, %@!", self.name]
```

zu arbeiten, verwenden wir Folgendes:

```
[NSString stringWithFormat:@"Hello, %@!", self.name]
```

Diese beiden Versionen sind *nicht* genau gleich. Sehen wir uns die Regeln an.

Wenn Sie den String mit der ersten Version erzeugen, gehört er Ihnen, und Sie sind für seine Freigabe verantwortlich. Wenn Sie `alloc` verwenden, erhöhen Sie den Referenzzähler um 1 und müssen den String wieder freigeben, indem Sie entweder `release` aufrufen oder den Autorelease-Mechanismus nutzen.

Was sagen nun die Regeln für ein Objekt, das mit der zweiten Methode erzeugt wurde? Wir haben `alloc` oder `new` nicht explizit genutzt, um den String zu erzeugen, und wir haben ihn auch nicht mit `retain` oder `copy` festgehalten. Darum gehört er uns nicht.

`stringWithFormat:` wird als Bequemlichkeitskonstruktor bezeichnet, weil er ein Objekt erzeugt und initialisiert und dann bereits „autoreleased“ an uns zurückgibt. Mit anderen Worten ist

```
[NSString stringWithFormat:@"Hello, %@!", self.name]
```

identisch mit

```
[[[NSString alloc] initWithFormat:@"Hello, %@!", self.name] autorelease]
```

6.13 Übung: Erzeugen und Nutzen eines Bequemlichkeitskonstruktors

Entwickeln Sie Ihren eigenen Bequemlichkeitskonstruktor für die `Greeter`-Klasse `greeterWithName:`. Sie müssen sie in `Greeter.h` deklarieren und in `Greeter.m` implementieren.

Refaktorieren Sie die Methode `applicationDidFinishLaunching:` in `FlashlightAppDelegate.m` so, dass sie diese Methode nutzt, um eine Instanz der `Greeter`-Klasse zu erzeugen. Nachdem Sie den String in der Konsole ausgegeben haben, wird der `Greeter` autoreleased, und die `Greeter`-Methode `dealloc` wird aufgerufen.

6.14 Lösung: Erzeugen und Nutzen eines Bequemlichkeitskonstruktors

Wir beginnen mit der Deklaration der Klassenmethode in `Greeter.h`:

Memory/Flashlight11/Classes/Greeter.h

```
+(id) greeterWithName:(NSString *) newName;
```

Wir implementieren die Methode, indem wir eine Autorelease-Instanz der `Greeter`-Klasse erzeugen und zurückgeben:

Memory/Flashlight11/Classes/Greeter.m

```
+(id) greeterWithName:(NSString *) newName {
    return [[[Greeter alloc] initWithName:newName] autorelease];
}
```

Die hervorgehobene Zeile zeigt, wie die Bequemlichkeitsmethode aufgerufen wird. Denken Sie daran, dass es eine Klassenmethode ist, dass Sie sie also für `Greeter` aufrufen und nicht für eine Instanz.

Memory/Flashlight11/Classes/FlashlightAppDelegate.m

```
#import "FlashlightAppDelegate.h"
#import "Greeter.h"

@implementation FlashlightAppDelegate
@synthesize window;

-(void)applicationDidFinishLaunching:(UIApplication *)application {
►    Greeter *host = [Greeter greeterWithName:@"Maggie"];
    NSLog(@"Greeter %@", host);
}
-(void) dealloc {
    [window release];
    [super dealloc];
}
@end
```

Wenn Sie sich die endgültige Version im Codedownload ansehen, werden Sie merken, dass ich auch eine `dealloc`-Methode in `FlashlightAppDelegate.m` eingefügt habe, um die `window`-Variable aufzuräumen. Das `dealloc` wird in der ursprünglichen Projektvorlage für Sie erzeugt, doch wir haben sie entfernt, um Speicherlecks zu erzeugen. Wenn Sie *Build & Analyze* anklicken, werden Sie sehen, das wir all unsere Speicherprobleme behoben haben.

In diesem Kapitel konnten Sie sich die beiden grundlegenden Typen von Speicherfehlern ansehen. Sie haben gelernt, wie man das Reference Counting bei iPhone OS–basierten Apps nutzt und dass man die Garbage Collection für Mac OS X-basierte Anwendungen (ab Leopard) aktivieren sollte. Sie haben auch erfahren, wie man den Clang Static Analyzer, Instruments, Zombies und Logging nutzt, um solche Lecks aufzuspüren.

Am Anfang dieses Buches haben wir mit einem funktionierenden Browser begonnen, den wir ohne eine einzige Zeile Code entwickelten. Nun haben wir eine Reihe von Kapiteln damit verbracht, uns die Grundlagen der Arbeit mit Objective-C und Cocoa anzusehen. Nun ist es an der Zeit, diese beiden Welten zu verbinden. Bevor wir uns von unserem iPhone-Beispiel abwenden und neue Projekte für Mac OS X angehen, möchte ich daran erinnern, die Garbage Collection zu aktivieren.

Kapitel 7

Outlets und Aktionen

Bei der Cocoa-Programmierung wird die Anwendungslogik per Model-View-Controller (MVC) vom „Look and Feel“ getrennt. Im Modell bauen wir die Anwendungslogik in Xcode mit Objective-C auf. Wie man den View im Interface Builder erzeugt, haben Sie ja bereits gesehen.

Der Controller bildet die Brücke zwischen Modell und View. Wenn der Benutzer einen Button anklickt, etwas in ein Textfeld einträgt oder etwas anderes am View verändert, reagiert der Controller häufig mit dem Senden einer Nachricht an das Modell. Ändert sich umgekehrt das Modell, aktualisiert der Controller den View, damit diese Änderungen für den Benutzer sichtbar werden.

Der Controller steht also mit jeweils einem Bein in beiden Welten. Es gibt eine Klassendatei, in der Sie Methoden entwickeln und Nachrichten an das Modell oder den View senden. Wir werden eine Instanz der Controller-Klasse im Interface Builder erzeugen. Das liefert uns eine visuelle Repräsentation des Controllers im Nib, mit dem Sie den Controller-Code mit den im IB erzeugten visuellen Komponenten verknüpfen. Es existieren sozusagen zwei Versionen des Controllers: eine reale im Programmcode und eine virtuelle im IB.

Wir werden im nächsten Kapitel einen Controller für unseren einfachen Browser entwickeln. In diesem Kapitel sehen wir uns an, wie die im Interface Builder erzeugten GUI-Elemente mit Ihrem Code kommunizieren und wie Ihr Code Änderungen an der GUI vornehmen kann. Sobald Sie ein Gefühl für diese Aktionen und Outlets haben, werden Sie sich ständig dabei ertappen, dass Sie sie benutzen.[1]

1 Wir beginnen zwar mit einer einfachen betrachtung von Outlets und Aktionen, werden aber im weiteren Verlauf deutlich anspruchsvollere Einsatzgebiete kennenlernen.

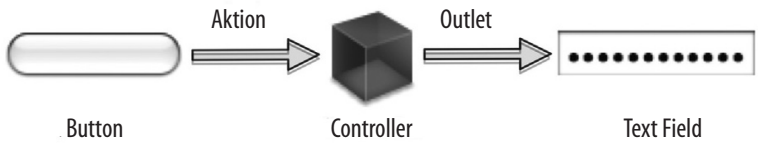
7.1 Das große Ganze

Stellen Sie sich ein Fenster mit einem Button und einem Textfeld vor. Wenn der Benutzer den Button anklickt, geben wir „Hello, World!“ im Textfeld aus. Wir benötigen eine Möglichkeit, um unseren Button mit Code zu verknüpfen, der beim Klick auf den Button eine Methode in unserem Code aufruft, die die Arbeit für uns erledigt. Wir benötigen außerdem eine Verbindung zum Textfeld, um dessen Text mit „Hello, World!“ festlegen zu können.

Es gibt grundsätzlich zwei Möglichkeiten für den Controller, die Verbindung mit den UI-Elementen herzustellen.

- *Aktionen*: Controller-Methoden, die benutzt werden, wenn ein Element (z. B. ein Button) eine vom Controller durchgeführte Aktion initiieren möchte.
- *Outlets*: Instanzvariablen des Controllers, die auf die UI-Elemente verweisen, an die der Controller Nachrichten senden muss.

Aktionen und Outlets (engl. „Steckdosen“) wurden speziell für Verbindungen entworfen, die im Interface Builder erzeugt wurden. Hier der grundlegende Kontrollfluss:



Klickt der Benutzer einen Button an, wird eine Nachricht an ein festgelegtes Ziel gesendet, um eine bestimmte Aktion zu initiieren. Sie bauen diese Aktion in einem Controller auf und stellen die Verbindung im Interface Builder her. Die Aktion ist einfach eine Methode, die aufgerufen wird, wenn der Button angeklickt wird.

Manchmal muss der Controller mit einem Objekt kommunizieren, das Sie im IB angelegt haben. Eine Möglichkeit besteht darin, dem Controller ein Handle auf dieses Objekt bereitzustellen. Stellen Sie sich vor, dass unser Controller ein Outlet besitzt, das ein Textfeld darstellt. Mit anderen Worten besitzt der Controller ein Instanzvariable, die auf ein Textfeld zeigt. Genau wie bei einer Steckdose ist das Outlet die Stelle im Controller, an der die visuellen Elemente „eingesteckt“ werden.

7.2 Ein Outlet benutzen

Legen Sie ein neues Projekt an. Verwenden Sie die Cocoa Application-Vorlage, deaktivieren Sie alle Checkboxen und nennen Sie es *HelloWorldPro*.

Fügen Sie die folgende Zeile in `HelloWorldProAppDelegate.m` ein:

Outlets/HelloWorldPro1/HelloWorldProAppDelegate.m

```
#import "HelloWorldProAppDelegate.h"

@implementation HelloWorldProAppDelegate

@synthesize window;
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
►    self.window.backgroundColor = [NSColor greenColor];
}
@end
```

Klicken Sie auf *Build & Run*, und das Fenster sollte mit einem grünen Hintergrund erscheinen. Wie ist das passiert? Wir haben eine Nachricht an die `window`-Eigenschaft gesendet, um die Hintergrundfarbe zu ändern. Wie ist diese Nachricht an das `NSWindow` gelangt, die Teil unserer *.nib*-Datei ist?

Damit das passiert, sind zwei wichtige Schritte nötig. Sehen Sie sich zuerst die Header-Datei der Klasse `HelloWorldProAppDelegate` an:

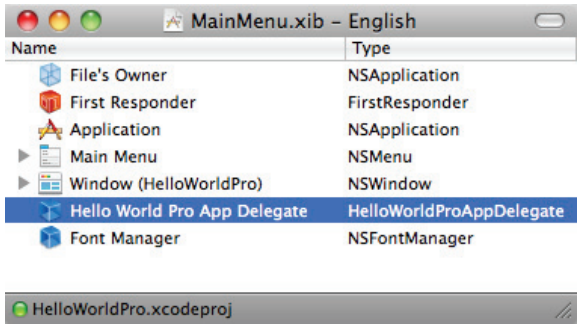
Outlets/HelloWorldPro1/HelloWorldProAppDelegate.h

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

@interface HelloWorldProAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
}
► @property (assign) IBOutlet NSWindow *window;
@end
```

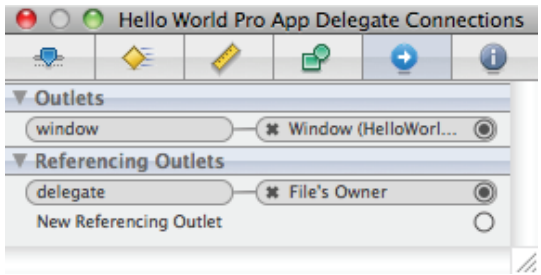
Sehen Sie das Schlüsselwort `IBOutlet` in der `@property`-Deklaration für `window`? Es weist den Interface Builder an, `window` in die Liste der Outlets für jede Instanz von `HelloWorldProAppDelegate` aufzunehmen.

Klicken Sie `MainMenu.xib` doppelt an (Sie finden die Datei unter *Resources*). Wenn das Nib im Interface Builder geöffnet wurde, sehen Sie sich das Dokumentenfenster an und wählen die Instanz der `HelloWorldProAppDelegate`-Klasse aus.[2]

2 Ist das Dokumentenfenster nicht sichtbar, können Sie es jederzeit mit ⌘ 0 oder `Window >Document` nach vorne holen.



Öffnen Sie den Connections Inspector (`Tools > Connections inspector`) und Sie erkennen, dass das `window`-Outlet bereits mit dem `NSWindow`-Objekt im selben Nib verknüpft wurde.

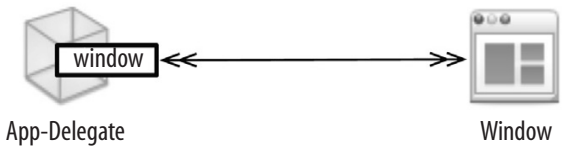


Wenn Sie wollen, können Sie sich auch die Verbindungen für das `NSWindow`-Objekt ansehen. Sie werden erkennen, dass `window` als referenzierendes Outlet aufgeführt ist.

Nun können Sie sich die Abfolge der Ereignisse vorstellen. Wir besitzen eine Variable und eine Eigenschaft `window` in der Klasse `HelloWorldProAppDelegate`. Wir besitzen ein Nib mit einer Instanz von `HelloWorldProAppDelegate` und einer Instanz von `NSWindow`.

Die `window`-Eigenschaft versteckt sich in `HelloWorldProAppDelegate`. Sie haben im vorigen Kapitel gesehen, wie wir auf diese Eigenschaft mit Code zugreifen können, aber wie verbinden wir sie im Interface Builder? Der erste Schritt besteht darin, diese Eigenschaft als `IBOutlet` zu kennzeichnen. Ich stelle mir dieses Outlet als Handle auf das Objekt vor, das im Interface Builder sichtbar ist und das ich verwenden kann, um die Variable mit anderen Objekten im selben Nib zu verknüpfen.

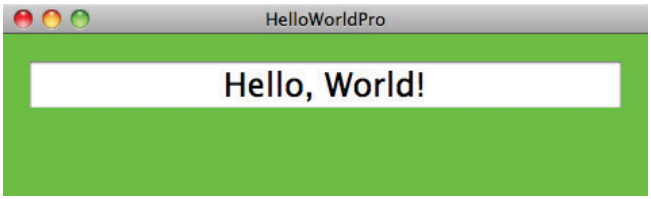
An diesem Punkt verknüpft nichts die `window`-Variable mit dem `NSWindow`-Objekt. Daher nutzen wir den Connections Inspector, um die beiden zu verbinden.



Nun sind die beiden verbunden. Wenn wir jetzt im Programmcode Nachrichten an die `window`-Variable senden, dann senden wir diese an das Objekt, mit dem `window` in der *.nib*-Datei verknüpft ist. Die `window`-Variable ist ein Proxy für das eigentliche Fenster. Wenn wir also in der `window`-Variablen die Hintergrundfarbe auf grün setzen, wird das `NSWindow`-Objekt grün.

7.3 Übung: Ein Outlet erzeugen und benutzen

Das Ziel dieser Übung ist der Einsatz von Outlets, um Folgendes zu erreichen:



Im Interface Builder ziehen Sie ein Textfeld aus der Library in das Fenster und positionieren es den Richtlinien entsprechend am oberen Rand. Zentrieren Sie den Text im Textfeld und passen Sie die Schriftgröße mithilfe des Attributes Inspector an. Speichern Sie Ihre Arbeit.

Fügen Sie eine neue Eigenschaft für das Textfeld in der Header-Datei von `HelloWorldProAppDelegate` ein und deklarieren Sie sie als Outlet. Verwenden Sie das Outlet, um „Hello, World!“ nach dem Start der Anwendung auszugeben.

Im Interface Builder verbinden Sie das Outlet mit dem Textfeld.

Klicken Sie *Build & Run* an, und Sie sollten im Textfeld die Nachricht „Hello, World!“ vor einem grünen Hintergrund erkennen können.

7.4 Lösung: Ein Outlet erzeugen und benutzen

Der Einsatz von Xode und Interface Builder folgt einem bestimmten Rhythmus. Sie wechseln ständig zwischen diesen beiden hin und her. Sie sollten keine Schwierigkeiten haben, im Interface Builder ein Textfeld in das Fenster zu ziehen und zu positionieren. Bevor Sie wieder zu Xcode wechseln, sehen Sie sich den Connections Inspector für `HelloWorldProAppDelegate` an. Es sollte genau aussehen wie vorher, also mit einem einzelnen bereits verbundenen Outlet.

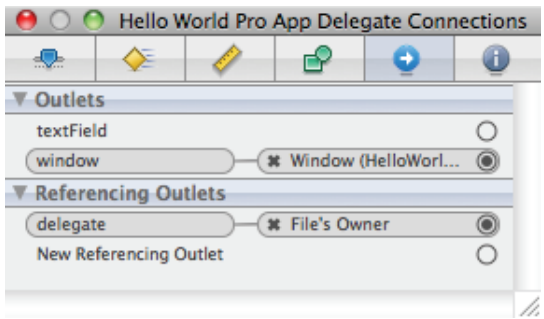
Der nächste Teil ist ziemlich cool. Fügen Sie die beiden hervorgehobenen Zeilen in Ihre Header-Datei ein:

Outlets/HelloWorldPro2/HelloWorldProAppDelegate.h

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

@interface HelloWorldProAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
►   NSTextField *textField;
}

@property (assign) IBOutlet NSWindow *window;
► @property (assign) IBOutlet NSTextField *textField;
@end
```

Sie haben eine Instanzvariable namens `textField` in `HelloWorldProAppDelegate` eingefügt und die entsprechende Eigenschaft deklariert. Für unseren Zweck noch wichtiger ist aber, dass Sie `textField` als Outlet markiert haben. Speichern Sie die Header-Datei und wechseln Sie wieder in den Connections Inspector im Interface Builder.

Sehen Sie das?



Ein neues Outlet ist erschienen. Das zaubert mir immer wieder ein Lächeln auf die Lippen. Wir haben ein Outlet in das öffentliche Interface für `HelloWorldProAppDelegate` eingefügt, um der Welt mitzuteilen, dass diese Eigenschaft verfügbar ist. Der Interface Builder hat darauf reagiert, weil Sie das Outlet als `IBOutlet` gekennzeichnet haben.

Klicken Sie den Kreis rechts neben dem `textField`-Outlet an und ziehen Sie ihn über das Textfeld innerhalb des Fensters. Lassen Sie die Maustaste los, und die neue Verbindung ist hergestellt. Speichern Sie Ihre Arbeit. Wenn Sie wollen, können Sie den Interface Builder nun vorerst beenden.

In Xcode müssen Sie nun zwei kleine Änderungen an der Implementierungsdatei vornehmen. Sie müssen die Akzessoren für `textField` synthetisieren, und Sie möchten „Hello, World!“ in diesem Feld ausgeben. Sie kennen diese beiden Schritte bereits.

Outlets/HelloWorldPro2/HelloWorldProAppDelegate.m

```
#import "HelloWorldProAppDelegate.h"
@implementation HelloWorldProAppDelegate

► @synthesize window, textField;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    self.window.backgroundColor = [NSColor greenColor];
►   [self.textField setStringValue:@"Hello, World!"];
}
@end
```

7.5 Eine Aktion deklarieren

Im Interface Builder fügen Sie einen Push-Button in Ihr Fenster ein und ändern seinen Titel in *Personalize*. Wird der Button angeklickt, ändern wir die Begrüßung von „Hello, World!“ in etwas Persönlicheres.

Wir müssen eine Methode in Xcode deklarieren und implementieren, die beim Klick auf den Button aufgerufen wird. Der erste Schritt besteht in der Deklaration der Methode in der richtigen Syntax. Für alle Cocoa-Anwendungen für Mac OS X müssen Ihre Aktionen wie folgt deklariert werden:

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

Das ist das Format, das Sie schon für die `goBack:`-Methode des *WebView* gesehen haben. Die Methode muss eine Instanzmethode sein, die einen einzelnen Parameter vom Typ `id` erwartet. Der Konvention entsprechend nennen wir den Parameter `sender`.

Joe fragt...

Wo finde ich das Outlet, das ich gerade im Code eingefügt habe?

Wenn Sie ein Outlet in eine Klasse namens `NSFoo` eingefügt haben, finden Sie das Outlet im IB bei jedem Objekt vom Typ `NSFoo`. Viele Interface Builder-Neulinge beschweren sich, dass sie ein Outlet in die Klasse aufgenommen haben, es aber nicht im IB erscheint. Es wird ihnen noch in Fleisch und Blut übergehen, doch üblicherweise liegt es an einem von zwei Gründen, wenn etwas schiefgeht:

- Stellen Sie sicher, dass Sie die Header-Datei in Xcode gespeichert haben. Noch zu sichernde Dateien erscheinen im *Groups & Files*-Panel leicht angegraut.
- Stellen Sie sicher, dass Sie nach dem richtigen Objekt suchen. In unserem Beispiel haben wir das Outlet in die Quelldatei `HelloWorldProAppDelegate.h` eingefügt; wählen Sie also im Interface Builder das `HelloWorldAppDelegate`-Objekt und sehen im Connections Inspector nach.

Wenn Ihnen diese Schritte offensichtlich erscheinen, umso besser: Sie haben die Aufgabenteilung zwischen Xcode und Interface Builder verstanden.

Der Rückgabewert ist vom Typ `IBAction`. Dieser Rückgabetyp signalisiert zweierlei. Zum einen erkennt der Interface Builder, dass es sich bei dieser Methode um eine Aktion handelt, und zeigt diese entsprechend im Connections Inspector an. Zum anderen ist `IBAction per typedef` als `void` deklariert, `IBAction` liefert also nichts zurück.[3]

Wenn Sie Anwendungen für das iPhone oder das iPod touch entwickeln, können Sie übrigens noch zwei weitere Varianten verwenden. Sie können einerseits eine Aktion ohne Argumente entwickeln. Die folgende Signatur funktioniert bei Cocoa Touch ausgezeichnet:[4]

```
-(IBAction) actionSelector
```

Diese Version ist immer meine erste Wahl, wenn ich mit Cocoa Touch arbeite. Würde mir diese Variante ohne Argument bei Cocoa zur Ver-

3 Der Rückgabetyp `IBAction` reicht allein nicht aus, um die Methode im Interface Builder erscheinen zu lassen. IB ist schlau genug, um sicherzustellen, dass sie nur einen einzigen Parameter vom Typ `id` verwendet.

4 Cocoa Touch nennen wir die Cocoa-APIs für das iPhone OS.

fügung stehen, wäre sie auch da meine erste Wahl. Für den Fall, dass ich den `sender` mitübergeben muss, schätze ich die Möglichkeit, es tun zu können. Doch häufig muss ich mit dem *sender* nicht kommunizieren und muss nichts über ihn wissen. Das würde ich in meinem Code deutlich machen, indem ich die Version ohne Argument nutzen würde.

Andererseits muss ich manchmal mehr wissen, als mir die Übergabe des `sender` verrät. Manchmal möchte ich etwas über das Ereignis wissen, das die Nachricht ausgelöst hat. Wenn Sie eine Cocoa Touch-Anwendung entwickeln, können Sie die folgende Signatur für eine `IBAction` verwenden:

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

Momentan müssen wir für Desktopanwendungen die Ein-Argument-Form zur Deklaration einer `IBAction` verwenden. Bei Cocoa Touch-Anwendungen können wir auch die Varianten ohne oder mit zwei Argumenten nutzen.

Zurück zu unserem Beispiel. Fügen Sie die hervorgehobene Deklaration in die Header-Datei ein und speichern Sie Ihre Arbeit:

Outlets/HelloWorldPro3/HelloWorldProAppDelegate.h

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

@interface HelloWorldProAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
    NSTextField *textField;
}
@property (assign) IBOutlet NSWindow *window;
@property (assign) IBOutlet NSTextField *textField;

► -(IBAction) changeGreeting:(id)sender;
@end
```

7.6 Die Aktion verknüpfen und implementieren

Sie haben den Button angelegt und die Aktion in der `HelloWorldProAppDelegate`-Header-Datei angelegt. Es könnte hilfreich sein, sich `IBAction` so vorzustellen, wie wir uns `IBOutlet` vorgestellt haben. Nehmen wir an, wir hätten die Methode in `HelloWorldProAppDelegate.h` so deklariert:

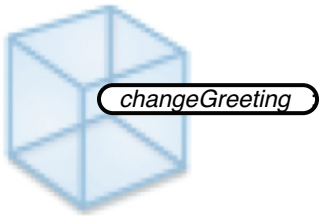
```
(void) changeGreeting
```

Diese Methode können wir zwar im Programmcode aufrufen, wir haben aber keine Möglichkeit, im Interface Builder eine Verknüpfung herzustellen:

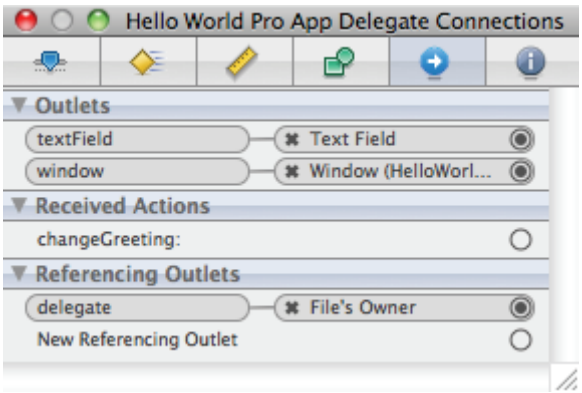
Ändern Sie die Signatur der Methode so ab, dass sie zu einer Aktion wird:

```
(IBAction) changeGreeting:(id) sender
```

Sie haben diese Methode nun offengelegt. Das bedeutet, dass Sie beispielsweise Buttons mit dieser Aktion verknüpfen können, damit die Methode ausgeführt wird, sobald der Button angeklickt wird.



Lassen Sie uns die deklarierte Aktion nutzen. Wechseln Sie zum Interface Builder und verknüpfen Sie die Aktion mit dem Button. Im IB wählen Sie das Objekt vom Typ `HelloWorldProAppDelegate` im Dokumentenfenster und sehen sich seine Verbindungen mit dem Connections Inspector an. Sie sollten `changeGreeting:` unter *Received Actions* sehen.



Klicken Sie den Kreis rechts neben `changeGreeting:` an und ziehen Sie ihn über den von Ihnen erzeugten Button. Lassen Sie die Maustaste los, und schon ist Ihr Outlet verknüpft. Sichern Sie Ihre Arbeit und beenden Sie den Interface Builder.

Wir implementieren die Methode so, dass sie die Hintergrundfarbe in rot ändert und den Benutzer persönlich grüßt. Wie können wir den Namen des Nutzers im Voraus kennen? Nun, das können wir natürlich nicht, aber wir können die Funktion `NSFullUserName()` nutzen, um den Namen zu ermitteln, den der Benutzer bei der Konfiguration seines Rechners angegeben hat:

Outlets/HelloWorldPro3/HelloWorldProAppDelegate.m

```
-(IBAction) changeGreeting:(id)sender {
     self.window.backgroundColor = [NSColor redColor];
    [self.textField setStringValue:
          [NSString stringWithFormat:@"Hello, %@!", NSFullUserName()]];
}
```

Wenn ich diese Anwendung auf meinem Laptop ausführe, sehe ich Folgendes:

7.7 Übung: Den Button verstecken

Mir gefällt der aktuelle Stand unserer Anwendung nicht. Nachdem der Benutzer *Personalize* angeklickt und das Textfeld die persönliche Begrüßung ausgegeben hat, sagt der Button immer noch „Personalize“, obwohl es für den Benutzer nichts mehr zu tun gibt.

Lassen Sie den Button verschwinden, nachdem der Benutzer die Ausgabe personalisiert hat.

7.8 Lösung: Den Button verstecken

Tendenziell werden Sie bei der Entwicklung von Cocoa-Anwendungen in Objective-C weniger Code schreiben, doch es wird Sie anfänglich mehr Zeit kosten, herauszufinden, welchen Code Sie schreiben müssen. Sie werden viel Zeit mit der Dokumentation und bei der Suche im Internet verbringen. Ein wichtiger Aspekt der Cocoa-Programmierung besteht darin, nicht mehr zu tun als man tun muss.

Zum Beispiel besteht hier unser Ziel darin, den Button zu verstecken. Wir wissen, dass wir dem Button eine Nachricht schicken müssen, doch wie machen wir das? Wir könnten ein Outlet für einen `NSButton` in `HelloWorldProAppDelegate.h` anlegen und im Nib das Outlet mit dem Button verknüpfen. Das wäre korrekt, aber unnötig.

Zumindest im Moment ist die `changeGreeting:`-Methode die einzige Stelle, an der wir auf den Button zugreifen müssen. Und Sie besitzen dort ein Handle auf diesen Button – den `sender`, der als Parameter übergeben wurde. Sie müssen dem Button nur noch die richtige Nachricht senden. Dazu fügen wir in unsere `changeGreeting:`-Implementierung die folgende Zeile ein:

Outlets/HelloWorldPro4/HelloWorldProAppDelegate.m

```
-(IBAction) changeGreeting:(id)sender {
    self.window.backgroundColor = [NSColor redColor];
    [self.textField setStringValue:
          [NSString stringWithFormat:@"Hello, %@!", NSFullUserName()]];
►   [sender setHidden:YES];
}
```

Wenn Sie sich damit wohler fühlen, können Sie den *sender* mit Typecast als `NSButton` angeben, damit deutlich wird, wem Sie die Nachricht senden:

```
[(NSButton *)sender setHidden:YES];
```

Woher wissen wir nun, welche Nachricht an den Button zu senden ist? Zuerst sehen Sie sich die Dokumentation für `NSButton` an und suchen nach etwas, das mit dem Verstecken eines Buttons zu tun hat. Leider finden Sie nichts. Also bewegen Sie sich zum Anfang der `NSButton`-Dokumentation und sehen, dass er von `NSControl : NSView : NSResponder : NSObject` erbt. Bewegen Sie sich im Vererbungsbaum nach oben. Sie sehen sich `NSControl` an, finden nichts und machen mit `NSView` weiter.

Sie gehen die Dokumentation durch, wo Sie die Überschrift *Hiding Views* finden, und eine der Methoden heißt `setHidden:`. Wir hätten uns auch dazu entscheiden können, den Button vollständig zu entfernen:

```
[sender removeFromSuperview];
```

Aus Sicht des Benutzers gibt es zwischen diesen beiden Ansätzen keinen Unterschied. Aus Sicht des Arbeitsspeichers gibt es aber einen sehr wichtigen Unterschied: Sobald wir den Button im Interface Builder in das Fenster gezogen haben, gehört der Button dem Superview des Buttons (also dem Content-View des Fensters), und der Referenzzähler wird um 1 erhöht. Wenn wir den Button verstecken, ist er immer noch vorhanden, er ist bloß nicht sichtbar. Wenn wir andererseits den Button aus dem Superview entfernen, gibt das einzige Objekt, dem der Button gehört, den Button frei – der Button selbst wird aus dem Speicher entfernt.

Keiner dieser Ansätze ist „der richtige“. Sie müssen in der jeweiligen Situation entscheiden, ob Sie den Button noch benötigen oder ob Sie ihn freigeben wollen.

7.9 Übung: Das Interface umschalten

Statt den Button zu verstecken, wollen wir ihn so einstellen, dass zwischen den beiden Views hin- und hergeschaltet wird. Beim Start der Anwendung enthält der Button den Text „Personalize“, der Hintergrund ist grün und im Textfeld steht „Hello, World!“

Wenn der Benutzer den *Personalize*-Button anklickt, soll der Button den Text „Generalize“ ausgeben, der Hintergrund soll rot werden und im Textfeld soll die personalisierte Begrüßung erscheinen.

7.10 Lösung: Das Interface umschalten

Es gibt viele Möglichkeiten, diese Aufgabe zu programmieren. Wir beginnen mit etwas Leichtem und sehen dann weiter. Ich füge eine boolesche Instanzvariable namens `isPersonalized` in die Header-Datei ein:

Outlets/HelloWorldPro5/HelloWorldProAppDelegate.h

```
NSWindow *window;
NSTextField *textField;
► BOOL isPersonalized;
```

Nun können wir die changeGreeting:-Methode so ändern, dass Button, Textfeld und Hintergrundfarbe basierend auf dem Wert von isPersonalized (YES oder NO) gesetzt werden. Ich habe isPersonalized in der Methode application-DidFinishLaunching: initialisiert. Der BOOLesche Wert wird standardmäßig mit NO initialisiert, aber ich finde es hilfreich, wenn man das explizit dazusagt:

Outlets/HelloWorldPro5/HelloWorldProAppDelegate.m

```
#import "HelloWorldProAppDelegate.h"

@implementation HelloWorldProAppDelegate

@synthesize window, textField;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    self.window.backgroundColor = [NSColor greenColor];
    [self.textField setStringValue:@"Hello, World!"];
    isPersonalized = NO;
}

-(IBAction) changeGreeting:(id)sender {
    if (isPersonalized) {
        self.window.backgroundColor = [NSColor greenColor];
        [self.textField setStringValue:@"Hello, World!"];
        [sender setTitle:@"Personalize"];
        isPersonalized = NO;
    } else {
    self.window.backgroundColor = [NSColor redColor];
    [self.textField setStringValue:
          [NSString stringWithFormat:@"Hello, %@!", NSFullUserName()]];
        [sender setTitle:@"Generalize"];
        isPersonalized = YES;
    }
}
@end
```

Klicken Sie auf *Build & Run*, und die Anwendung wird sich wie gewünscht verhalten.

7.11 Ein weiteres Outlet hinzufügen

Mir gefällt der sich wiederholende Code in der Methode application-DidFinishLaunching: und im Wahr-Zweig der if-Anweisung von change-Greeting: nicht. Wir setzen die Hintergrundfarbe und den Inhalt des Textfelds zweimal auf den gleichen Wert. Ich möchte daher change-Greeting: aus applicationDidFinishLaunching: heraus aufrufen.

Erkennen Sie das Problem dabei?

Wer soll dabei der `sender` sein? Wie bekommt `changeGreeting:` ein Handle auf den Button, wenn es nicht als Reaktion auf einen Button-Klick aufgerufen wird?

Lassen Sie uns ein Outlet für den Button hinzufügen:

Outlets/HelloWorldPro6/HelloWorldProAppDelegate.h

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

@interface HelloWorldProAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
    NSTextField *textField;
►   NSButton *button;
    BOOL isPersonalized;
}

@property (assign) IBOutlet NSWindow *window;
@property (assign) IBOutlet NSTextField *textField;
► @property (assign) IBOutlet NSButton *button;

-(IBAction) changeGreeting:(id)sender;
@end
```

Verknüpfen Sie dieses Outlet im Interface Builder mit Ihrem Button. Wenn Sie wollen, können Sie den Titel des Buttons löschen, sodass er leer bleibt. Speichern Sie Ihre Arbeit.

In der Implementierungsdatei müssen Sie `button` synthetisieren. Dann können Sie den `sender` in `changeGreeting:` in `self.button` ändern. Zuerst verunstalten wir `applicationDidFinishLaunching:` etwas, bevor wir es gleich verbessern.

Outlets/HelloWorldPro6/HelloWorldProAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    self.window.backgroundColor = [NSColor greenColor];
    [self.textField setStringValue:@"Hello, World!"];
    [self.button setTitle:@"Personalize"];
    isPersonalized = NO;
}
```

Dieser Code entspricht nun dem Wahr-Teil der `if`-Anweisung. Das verlangt eine Refaktorierung. Wir führen zwei Hilfsmethoden (`personalize` und `generalize`) ein und nutzen diese in `applicationDidFinish-Launching:` und `changeGreeting:`.

Outlets/HelloWorldPro7/HelloWorldProAppDelegate.m

```
#import "HelloWorldProAppDelegate.h"

@implementation HelloWorldProAppDelegate

@synthesize window, textField, button;

-(void) personalize {
    self.window.backgroundColor = [NSColor redColor];
    [self.textField setStringValue:
    [NSString stringWithFormat:@"Hello, %@!", NSFullUserName()]];
    [self.button setTitle:@"Generalize"];
    isPersonalized = YES;
}
-(void) generalize {
    self.window.backgroundColor = [NSColor greenColor];
    [self.textField setStringValue:@"Hello, World!"];
    [self.button setTitle:@"Personalize"];
    isPersonalized = NO;
}
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    [self generalize];
}
-(IBAction) changeGreeting:(id)sender {
    if (isPersonalized) [self generalize];
    else [self personalize];
}
@end
```

Der Code ist eigentlich nicht kürzer, aber einfacher zu lesen. Im nächsten Abschnitt nutzen wir ein weiteres Feature von Objective-C, um eine weitere Änderung vorzunehmen.

7.12 Selektoren aus Strings erzeugen

Gehen wir einen Schritt zurück und sehen wir uns den Text des Buttons noch einmal genau an. Wenn dieser „Personalize“ lautet, rufen wir die Methode `personalize` auf, und wenn er „Generalize“ lautet, die Methode `generalize`.

Es wäre nett, wenn wir den Titel des Buttons nehmen, in Kleinbuchstaben umwandeln und als Name der aufzurufenden Methode verwenden könnten. Da sich der Wert zur Laufzeit ändert, nutzen wir die Funktion `NSSelectorFromString()` so:

```
NSSelectorFromString([[self.button title] lowercaseString]);
```

Das liefert uns einen Selektor zurück, was man sich als formalen Namen der Methode vorstellen kann. Wir rufen die Methode mit diesem Namen auf, indem wir die Nachricht `performSelector:` an `self` senden und dabei den gerade erzeugten Selektor als Argument übergeben.

Wir ändern `changeGreeting:` also wie folgt:

Outlets/HelloWorldPro8/HelloWorldProAppDelegate.m

```
-(IBAction) changeGreeting:(id)sender {
    [self performSelector:
        NSSelectorFromString([[self.button title] lowercaseString])];
}
```

Wo wir gerade dabei sind, können wir das BOOLesche `isPersonalized` gleich löschen, weil wir es nicht mehr benötigen. Ich möchte keineswegs jeden booleschen Wert und jede if-Anweisung ersetzen, aber ich möchte Ihnen eine Möglichkeit aufzeigen, wie Sie Selektoren nutzen können, ohne Entscheidungen treffen zu müssen.[5]

Wir haben unsere Reise mit einem Webbrowser begonnen, für den wir keinerlei Code schreiben mussten. In den bisherigen Kapiteln haben Sie viel über den Code und die Verbindungen gelernt, die für Sie erzeugt wurden. Im nächsten Kapitel kehren wir zu unserem Webbrowser zurück und entwickeln einen Controller, um die Funktionalität zu erweitern.

5 In diesem Fall habe ich die Entscheidung an das Label des Buttons gekoppelt. Das wird aber schwierig, wenn es um die Internationalisierung der Anwendung geht. Mein Ziel war, Ihnen `NSSelectorFromString()` vorzustellen. Sie werden robustere Möglichkeiten zur Nutzung dieser Technik kennenlernen.

Kapitel 8

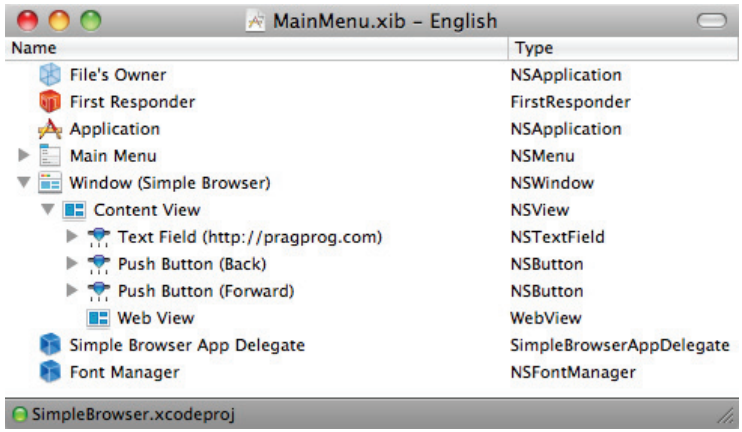
Einen Controller entwickeln

Sie können nicht alle Anforderungen einer Anwendung erfüllen, indem Sie im Interface Builder einfach Verbindungen zwischen den visuellen Elementen herstellen. Einerseits ist es faszinierend, wie einfach wir allein mit diesen visuellen Tools einen simplen Webbrowser aufbauen konnten (in Kapitel 2, *Vorhandenes nutzen*, auf Seite 11). Andererseits bleibt noch eine ganze Reihe von Wünschen offen. Es gibt einiges, was wir schlicht und einfach selbst programmieren müssen.

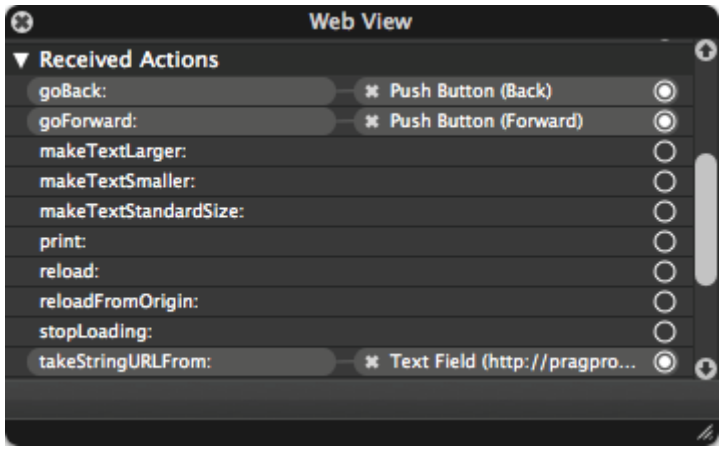
In diesem Kapitel wollen wir einen Controller für unser *SimpleBrowser*-Beispiel entwickeln. Um das Ganze einfach zu halten, verzichten wir auf ein Modell – wir benutzen nur einen View und einen Controller. Der wesentliche Punkt bei diesem Beispiel ist die Entwicklung einer neuen Klasse und deren Instanziierung zur Kommunikation mit Objekten, die im Interface Builder erzeugt wurden.

8.1 Wie wir Objekte erzeugt haben

Wir haben Objekte auf zwei unterschiedliche Arten erzeugt und verknüpft. Im *SimpleBrowser*-Beispiel wurden alle Objekte im Nib erzeugt. Wir haben zwei Buttons, ein Textfeld und einen Web-View in unser Fenster gezogen; sie schlossen sich dem `SimpleBrowserAppDelegate` und allen anderen im Nib erzeugten Objekten an.



Es gibt effektiv keinen Code für diese Anwendung. Alle Verbindungen zwischen den Objekten wurden im Interface Builder hergestellt. Das hier sind die Aktionen, die der Web-View empfängt: eine `goBack:`-Nachricht vom *Back*-Button, eine `goForward:`-Nachricht vom *Forward*-Button und die `takeStringURLFrom:`-Nachricht vom Textfeld.



Vergleichen Sie das mit der „Hallo, Welt!“-Anwendung aus Kapitel 4, *Klassen und Objekte*, auf Seite 51. Dort haben wir im Delegate für die Anwendung ein Textfeld im Programmcode erzeugt. Wir haben dann einen eigenen `Greeter` entwickelt. Wir haben ihn im Code des App-Delegates instanziiert, und die gesamte Kommunikation zwischen den erzeugten Objekten erfolgte ebenfalls über Programmcode.

In Kapitel 7, *Outlets und Aktionen*, auf Seite 117 haben Sie erfahren, wie Sie selbstentwickelten Code mit Objekten verknüpfen, die im Interface Builder erzeugt wurden. Wenn Sie aus Ihrem Code mit einem Widget

kommunizieren mussten, haben Sie in Ihrer Header-Datei ein Outlet für dieses Widget erzeugt und dieses Outlet dann im IB mit dem Widget verknüpft.[1] Sollte ein Widget eine von Ihnen entwickelte Methode anstoßen, haben Sie die Aktion in der Header-Datei deklariert und das Widget im Interface Builder mit der Aktion verknüpft.

Der wesentliche Punkt ist dabei, das es in Ihrem Nib ein Objekt des Typs geben muss, für den diese Outlets und Aktionen definiert werden. Im Augenblick bedeutet das, dass wir eine Instanz unserer Klasse im Interface Builder erzeugen.

Lassen Sie uns das etwas konkretisieren: Wir wollen eine Controller-Klasse entwickeln und dann eine entsprechende Instanz im Interface Builder erzeugen.

8.2 Eine Controller-Klasse entwickeln

Alle *Klassen* werden in Xcode entwickelt.

Öffnen Sie das *SimpleBrowser*-Projekt. In Xcode wählen Sie `File > New File...` oder ⌘N. Wählen Sie dann `Cocoa > Objective-C class`. Ich weiß, dass das nicht nach einer Controller-Klasse aussieht und dass es andere Optionen gibt, die das Wort *Controller* enthalten. Die wollen wir aber nicht. Was diese Klasse zu einem Controller macht, ist die Art und Weise, wie sie konfiguriert und verwendet wird.

Nennen Sie die Klasse `BrowserController` und stellen Sie sicher, dass die Checkboxen zur Generierung von `BrowserController.h` und für das Ziel *SimpleBrowser* aktiviert sind. Im Allgemeinen sollte es reichen, wenn Sie die Voreinstellungen übernehmen. Klicken Sie auf *Finish* und speichern Sie.

Unser nächster Schritt besteht darin, eine Instanz der Klasse zu erzeugen und ihr zu erlauben, mit den bereits erzeugten GUI-Elementen zu interagieren. Sie können `BrowserController` über in Xcode geschriebenen Programmcode instanziieren, oder so, wie wir GUI-Elemente wie `NSButton` im Interface Builder instanziiert haben.

Zwar werden Sie Ihre Klassen immer mit Xcode entwickeln, aber Sie haben gesehen, dass man sie im Programmcode oder mithilfe des Interface Builders instanziieren kann. Wir werden Objekte erzeugen, die in Xcode zum Modell gehören, weil sie nicht direkt mit den GUI-Elemen-

1 Ich verwende informell den Begriff „Widget“ für GUI-Elemente wie Buttons, Textfelder, und so weiter.

ten kommunizieren und auch nichts über sie wissen müssen. Wir werden Objekte entwickeln, die im Interface Builder Controller-Elemente darstellen, sodass wir Verbindungen zwischen den Controllern und den Objekten herstellen können, mit denen sie kommunizieren.

Zuerst wird Ihnen das ein wenig seltsam vorkommen. Schließlich erzeugen Sie die Instanz einer Klasse ohne visuelle Darstellung mithilfe eines Tools namens *Interface* Builder. Doch nach kurzer Zeit wird es sich ganz normal anfühlen.

8.3 Eine Instanz unseres Controllers in IB erzeugen

Wir wollen nun eine Instanz unserer `BrowserController`-Klasse im Interface Builder erzeugen.[2]

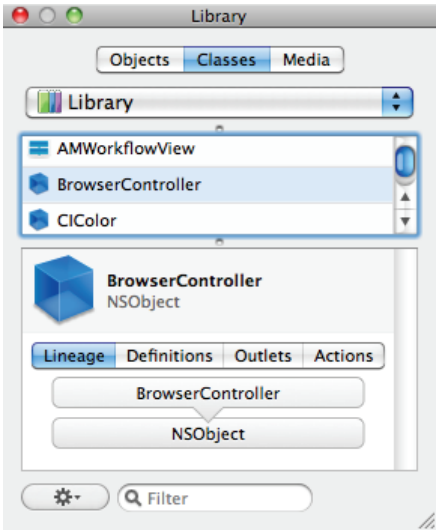
Als wir Instanzen unserer Buttons erzeugten, haben wir einfach in der Library nach einem `NSButton` gesucht, der so aussah, wie wir ihn wollten, und haben diesen dann in unser Fenster gezogen. Mit unserem `BrowserController` ist das nicht möglich, weil die Library des Interface Builders unsere `BrowserController`-Klasse nicht enthält – wir haben Sie gerade erst selbst entwickelt. Glücklicherweise haben Snow Leopard und Xcode 3.2 die Sache für uns sehr viel einfacher gemacht.[3]

Klicken Sie `MainMenu.xib` doppelt an, um die *.nib*-Datei im IB zu öffnen. Sie sind nicht am Window-View interessiert, weil es keine visuelle Repräsentation des Controllers gibt, die der Endanwender sehen könnte. Stattdessen öffnen Sie im Interface Builder über die Tastenkombination ⌘ 0 oder über `Window > Document` das Dokumentenfenster.

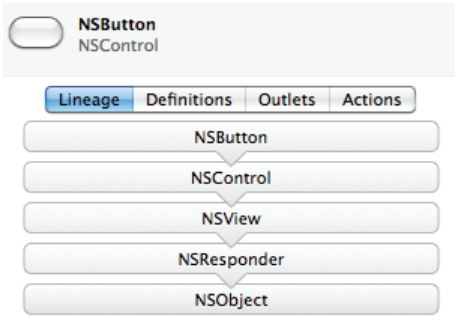
Der Interface Builder bietet uns eine Repräsentation unseres `BrowserController`-Objekts an. Wechseln Sie zur Library, wählen Sie diesmal den Reiter *Classes* und suchen Sie nach `BrowserController`.

2 Tatsächlich wird die Instanz erst erzeugt, wenn das Nib beim Start der Anwendung dearchiviert wird. An diesem Punkt können wir uns das Erzeugen der Instanz so vorstellen wie die Erzeugung eines Objekts im Programmcode über einen Aufruf wie `[[BrowserController alloc ] init ]`.

3 Wenn Sie mit einer älteren Version von Xcode arbeiten, müssen Sie die folgenden Anweisungen ändern. Ziehen Sie ein `NSObject` in das Dokumentenfenster und ändern Sie dann seinen Typ über den Identity Inspector.



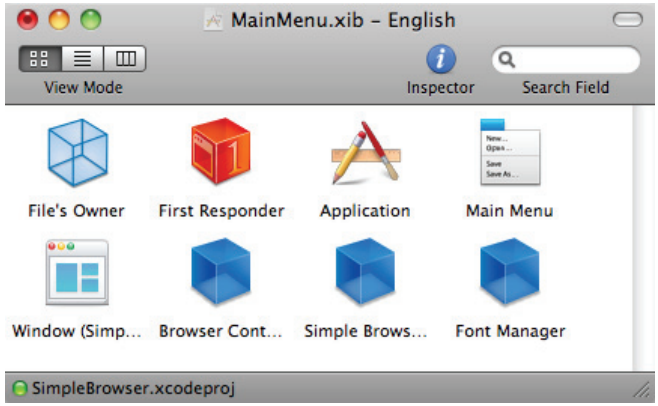
Die Vererbung (inheritance) der `BrowserController`-Klasse ist einfach, da sie `NSObject` direkt erweitert. Wenn Sie sich die Vererbung einer Klasse wie `NSButton` ansehen, erkennen Sie eine wesentlich tiefer reichende Hierarchie.



Neben der Vererbung können Sie auch die Outlets und Aktionen der Klasse inspizieren und nachsehen, wo die Klasse definiert ist. Sie werden das + und - in der unteren linken Ecke der *Outlet-* und *Action-*Reiter bemerken. Sie sollten im Interface Builder keinen Outlets und Aktionen hinzufügen oder entfernen. Es sieht zwar so aus, als würde das die Lage vereinfachen, aber im IB vorgenommene Änderungen werden von Xcode nicht wahrgenommen. Nehmen Sie die Änderungen im Code vor und überlassen Sie die Verarbeitung der Änderungen dem IB.

Zurück zu unserem Beispiel. Ziehen Sie den `BrowserController` in das Dokumentenfenster und lassen Sie ihn los. Glückwunsch! Sie haben soeben eine Instanz von `BrowserController` im Nib erzeugt. Hier sehen

Sie den Icon-View des Dokumentenfensters mit der neu hinzugefügten Instanz von `BrowserController`:



Ich bevorzuge die Listenansicht, wollte Ihnen aber (für den Fall, dass er Ihnen besser gefällt) auch diesen View zeigen.

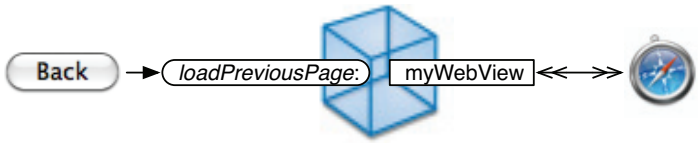
8.4 Ein Outlet und eine Aktion deklarieren

Im Moment ist der *Back*-Button mit der `goBack:`-Methode des Web-Views verknüpft. Nun wollen wir den `BrowserController` dazwischenschalten. Wir benötigen im Controller eine Methode zum Laden der vorherigen Webseite. Diese wird vom *Back*-Button aufgerufen und muss im Gegenzug die `goBack:`-Nachricht an den Web-View senden. Wir benötigen also ein Outlet für den Web-View und eine Aktion für die Methode.

In Xcode fügen Sie eine `IBAction` namens `loadPreviousPage:` in `BrowserController.h` ein. Sie müssen außerdem mithilfe einer Instanzvariablen und einer Eigenschaft ein `IBOutlet` namens `myWebView` einfügen.[4] Die `myWebView`-Variable ist ein Zeiger auf ein `WebView`-Objekt.

Sobald Sie das eingerichtet haben, wird beim Klick auf den *Back*-Button die `loadPreviousPage:`-Methode des `BrowserController`-Objekts aufgerufen. Die Methode `loadPrevious-Page:` ruft die `goBack:`-Methode für die `myWebView`-Variable auf. Da die `myWebView`-Variable auch ein Outlet ist, das mit dem `WebView`-Objekt verbunden ist, wird die `goBack:`-Methode für dieses `WebView`-Objekt aufgerufen. Mehr über Objekte und Aktionen erfahren Sie in Apples *Communicating with Objects* [App08c].

4 Wenn Sie mit einem 64-Bit-System arbeiten und für solche Systeme entwickeln, können Sie die Eigenschaft ohne eine Instanzvariable erzeugen.



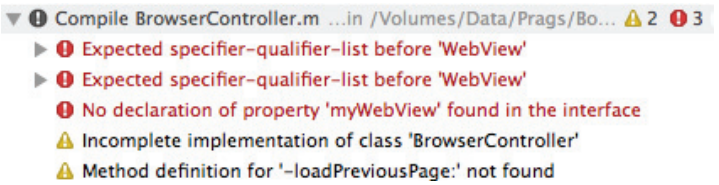
Denken Sie daran, `myWebView` in der Implementierungsdatei zu synthetisieren. Hier sehen Sie den Header:

CreatingAController/SimpleBrowser1/BrowserController.h

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

@interface BrowserController : NSObject {
    WebView *myWebView;
}
@property(assign) IBOutlet WebView *myWebView;

-(IBAction) loadPreviousPage: (id) sender;

@end
```

Sehen Sie irgendwelche Probleme, bevor Sie *Build & Run* anklicken? Wenn ja, dann korrigieren Sie sie. Wenn nicht, werde ich sie in einer Minute erklären. Klicken Sie *Build & Run* an, und Sie sehen die folgende Meldung: „Build failed (9 errors, 6 warnings)“. In der unteren rechten Ecke des Fenster sehen Sie das hier:

Um zu einem Fenster zu gelangen, das detaillierter beschreibt, was genau schiefgegangen ist, können Sie *Failed*, das gelbe Dreieck mit dem ! (eine Warnung) oder den roten Kreis mit dem ! (ein Fehler) anklicken. Die drei Fehler und zwei Warnungen werden dreimal wiederholt:



Sehen wir uns zuerst die Fehler an. Ihr `IBOutlet` verwendet die Klasse `WebView`, aber Ihr Programm weiß nichts von dieser `WebView`-Klasse. Sie müssen die entsprechende Header-Datei importieren. Am Anfang der `WebView`-Dokumentation können Sie sehen, dass sie Teil des `WebKit`-Frameworks und in `WebView.h` deklariert ist. Eine mögliche Lösung besteht darin, in `BrowserController.h` die folgende Zeile hinter dem Import der Cocoa-Header einzufügen:

Joe fragt...

Was ist der Unterschied zwischen einer Warnung und einem Fehler?

Ein *Fehler* liegt vor, wenn irgendetwas die Kompilierung des Codes verhindert, ist also ein schwerwiegender Softwarefehler. Ihr Code kann nicht kompiliert werden, also gibt es auch keine App, die man ausführen könnte. Es bleibt Ihnen nichts anderes übrig, als Fehler zu beheben.

Eine *Warnung* teilt Ihnen mit, dass es ein Problem geben könnte, das aber nicht so schwerwiegend ist, dass die Kompilierung des Codes beendet werden müsste. Das kann bedeuten, dass alles wunderbar funktioniert, es kann aber auch bedeuten, dass es zu einem Laufzeitfehler kommt. Eine Warnung kann beispielsweise der Hinweis während der Kompilierung sein, dass Sie eine Nachricht an ein Objekt senden, das keine Methode mit dieser Signatur deklariert hat. Das bedeutet nicht, dass das Objekt die Nachricht nicht verarbeiten könnte, es bedeutet nur, dass der Compiler nicht prüfen kann, ob das Objekt dazu in der Lage ist.

Es ist leicht, Warnungen zu ignorieren. Ihr Code wird kompiliert, warum sollten Sie sich also Sorgen machen? Einige erfahrene Entwickler finden, dass man es sich zu leicht macht, wenn man diese Warnungen ignoriert, und aktivieren daher die Option „Treat Warnings as Errors“, behandeln also Warnungen wie Fehler.

Wählen Sie `Project > Edit Project Settings` über das Menü, öffnen Sie den *Build*-Reiter und aktivieren Sie die Checkbox *Treat Warnings as Errors*. Suchen Sie nach dem Wort „treat“.

CreatingAController/SimpleBrowser2/BrowserController.h

```
#import <Cocoa/Cocoa.h>
► #import <WebKit/WebKit.h>
@interface BrowserController : NSObject {
    WebView *myWebView;
}
@property(assign) IBOutlet WebView *myWebView;
-(IBAction) loadPreviousPage: (id) sender;
@end
```

Klicken Sie *Build & Run* erneut an. Diesmal ist der Build erfolgreich. Es gibt immer noch zwei Warnungen, weil die Header-Datei eine Methode namens `loadPreviousPage` verspricht, die wir aber noch nicht imple-

mentiert haben. Doch obwohl wir unsere Arbeit noch nicht abgeschlossen haben, wird das Programm korrekt kompiliert und funktioniert wie erwartet. Beenden Sie die laufende Anwendung, indem Sie das rote *Tasks*-Symbol anklicken.

8.5 Vorwärtsdeklaration

Es gibt eine weitere Lösung für die Fehler aus dem vorigen Abschnitt: Sie können in der Header-Datei die `@class`-Direktive anstelle der `import`-Anweisung verwenden.

CreatingAController/SimpleBrowser3/BrowserController.h

```
#import <Cocoa/Cocoa.h>
► @class WebView;
@interface BrowserController : NSObject {
    WebView *myWebView;
}
@property(assign) IBOutlet WebView *myWebView;
-(IBAction) loadPreviousPage: (id) sender;
@end
```

In der Datei `BrowserController.h` muss der Compiler nur wissen, das `WebView` eine gültige Klasse ist. Sie muss nichts über die Klasse wissen. Die `@class`-Direktive macht genau das. Sie fügt nicht die ganze Header-Datei ein, die uns sagt, was ein Objekt vom Typ `WebView` alles kann – wir müssen das an dieser Stelle nicht wissen; sie garantiert uns nur, dass eine Klasse namens `WebView` existiert.

Wenn Sie eine `@class`-Direktive in einer Header-Datei verwenden, benötigen Sie sehr wahrscheinlich auch eine passende `import`-Anweisung in der Implementierungsdatei. Fügen Sie diese also am Anfang der Implementierungsdatei ein. Und wo Sie gerade dabei sind, können Sie auch die `loadPreviousPage:`-Aktion einfügen.

CreatingAController/SimpleBrowser3/BrowserController.m

```
#import "BrowserController.h"
► #import <WebKit/WebKit.h>

@implementation BrowserController

@synthesize myWebView;

-(IBAction) loadPreviousPage: (id) sender{
    NSLog(@"loadPreviousPage:");
}

@end
```

Klicken Sie auf *Build & Run*. Es sollte nun weder Warnungen noch Fehler geben, und der Browser sollte wie erwartet funktionieren.

8.6 Den Controller verknüpfen

Wechseln Sie in den Interface Builder. Im Dokumentenfenster klicken Sie das `BrowserController`-Objekt an und öffnen den Connections Inspector.

Unter *Outlets* finden Sie nun `myWebView` und unter *Received Actions* sehen Sie `loadPreviousPage:`. Verbinden Sie die Aktion `loadPreviousPage:` mit dem *Back*-Button. Er sollte jetzt nur mit dem Controller verbunden sein, nicht direkt mit dem Web-View. Verknüpfen Sie außerdem das `myWebView`-Outlet mit dem Web-View.

Speichern Sie und klicken Sie auf *Build & Run*. Geben Sie ein paar URLs ein. Probieren Sie den *Back*-Button aus. Wenn Sie ihn anklicken, wird die Nachricht `loadPreviousPage:` an das `BrowserController`-Objekt gesendet. Im Moment wird dabei nur der Methodenname in der Konsole ausgegeben.

8.7 Das Laden der vorigen Seite implementieren

Zurück in Xcode gibt es nicht viel Code zu entwickeln. Wenn der *Back*-Button die `loadPreviousPage:`-Nachricht an das `BrowserController`-Objekt sendet, schickt es seinerseits nur die `goBack`-Nachricht an `myWebView`.

Die einzige Entscheidung, die wir treffen müssen, ist die nach dem Sender, den wir an den Web-View übergeben. Wir können `self` senden oder die Identität des Objekts übergeben, das `loadPreviousPage:` aufgerufen hat. In unserem Fall spielt das keine Rolle. Ich habe mich für die zweite Variante entschieden und übergebe den *Back*-Button als Sender.

CreatingAController/SimpleBrowser4/BrowserController.m

```
-(IBAction) loadPreviousPage: (id) sender{
►    [self.myWebView goBack:sender];
}
```

Das war's! Klicken Sie *Build & Run* an – Sie besitzen wieder einen funktionierenden *Back*-Button.

8.8 Übung: Den Controller fertigstellen

Im `BrowserController` fügen Sie eine weitere Methode namens `loadNextPage:` ein und nutzen diese, um den *Foward*-Button über den Controller zu steuern.

Sobald das funktioniert, fügen Sie eine letzte Aktion in den `BrowserController` ein. Nennen Sie sie `loadURLFrom:` und benutzen Sie sie, um den URL-Eintrag aus dem Textfeld durch den Controller laufen zu lassen. Sie wissen ja noch, dass das Textfeld eine Nachricht an den Controller sendet, wenn die *Enter*-Taste gedrückt wird. Der Web-View muss dann eine Nachricht an das Textfeld senden, um den Stringwert des Feldes zu bestimmen. Sie können das auf unterschiedliche Weise lösen. Sie könnten z. B. versucht sein, ein Outlet für das Textfeld einzubinden. In diesem Fall ist das aber nicht nötig: Nutzen Sie den `sender`, um mit dem Textfeld zu kommunizieren.

8.9 Lösung: Den Controller fertigstellen

Die erste Hälfte der Übung entspricht dem ersten Schritt, den wir zusammen erledigt haben. Sie müssen drei Änderungen vornehmen.

Zuerst müssen Sie mit Xcode eine Aktion in der Header-Datei `BrowserController.h` einfügen und dann speichern:

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

Danach müssen Sie wieder in den Interface Builder wechseln und den `BrowserController` auswählen. Im Connections Inspector ziehen Sie den Kreis rechts neben `load-NextPage:` über den *Forward*-Button, um die Verbindung herzustellen. Speichern Sie.

Abschließend müssen Sie die Methode implementieren. Ein Blick auf die Verbindungen des Web-Views zeigt uns, dass wir die Methode `goForward:` aufrufen müssen. In Xcode erweitern wir anschließend `BrowserController.m` um die folgende Methode:

CreatingAController/SimpleBrowser5/BrowserController.m

```
-(IBAction) loadNextPage: (id) sender{
    [self.myWebView goForward:sender];
}
```

Speichern Sie und klicken Sie auf *Build & Run* – Sie sollten über funktionierende *Back*- und *Forward*-Buttons verfügen. Geben Sie ein paar URLs ein, und Sie sollten sich über die Buttons in der Liste vor- und zurückbewegen können.

Bei der zweiten Hälfte der Übung folgen wir drei ähnlichen Schritten. Zuerst erweitern wir die Header-Datei um eine Aktion namens `loadURLFrom:` und speichern die Datei ab:

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

Wir wählen den `BrowserController` im Dokumentenfenster im IB und öffnen den Connections Inspector.

Wir stellen die Verbindung her, indem wir den Kreis rechts neben `loadURLFrom:` in das Textfeld ziehen. Das Textfeld ist bereits so konfiguriert, dass es eine Aktion an das Ziel sendet, sobald der Benutzer die *Enter*-Taste drückt. Sie können sicherstellen, dass der `Action`-Wert immer noch in dieser Form gesetzt ist, indem Sie sich das Textfeld mit dem Attributes Inspector ansehen. Speichern Sie ab. In Xcode implementieren Sie die Methode wie folgt:

CreatingAController/SimpleBrowser5/BrowserController.m

```
-(IBAction) loadURLFrom: (id) sender{
    [self.myWebView takeStringURLFrom:sender];
}
```

8.10 awakeFromNib

In Kapitel 4, *Klassen und Objekte*, auf Seite 51 haben wir eine Instanz von `Greeter` über die folgende Kombination instanziiert:

```
[[Greeter alloc] initWithName:@"Maggie"];
```

Wir waren in der Lage, in der `initWithName:`-Methode unsere Variablen zu initialisieren und die notwendigen Anpassungen vorzunehmen. Eine vergleichbare Methode haben wir im `BrowserController` nicht. Er wird initialisiert, wenn der Nib geladen wird, und nicht durch einen expiziten Aufruf von `alloc` und irgendeine Form von `init`.

Ich werde später noch detaillierter darauf eingehen, was genau passiert, wenn ein Nib dearchiviert und geladen wird. Im Moment wollen wir es dabei belassen, dass beim Start der Anwendung der Objektgraph dearchiviert und das Nib rekonstruiert wird. Die Objekte werden erzeugt und die Verbindungen zwischen ihnen werden aufgebaut. Als Nächstes wird, bevor der Anwender etwas zu sehen bekommt, eine `awakeFromNib`-Nachricht an alle Objekte gesendet, die über diese Methode verfügen.[5] Fügen Sie diese Methode einfach in jede Datei ein, die nach der Initialisierung weitere Aufgaben übernehmen muss:

```
-(void) awakeFromNib {
}
```

Ich möchte beispielsweise, dass der Browser beim Start unsere Standardwebseite lädt. Im Moment muss der Benutzer noch das Textfeld anklicken und *Enter* drücken, damit die Seite geladen wird. Also setze ich in `awakeFromNib` das Textfeld auf den Stringwert `http://pragprog.com` und lasse den Web-View diese Seite darstellen.

Das bedeutet, dass ich in der Lage sein muss, mit dem Textfeld im Body von `awakeFromNib` zu interagieren. Wir müssen für das `NSTextField` ein Outlet in der Header-Datei einfügen. Wir nennen es `address`.

CreatingAController/SimpleBrowser6/BrowserController.h

```
#import <Cocoa/Cocoa.h>
@class WebView;
@interface BrowserController : NSObject {
    WebView *myWebView;
►   NSTextField *address;
}
@property(assign) IBOutlet WebView *myWebView;
► @property(assign) IBOutlet NSTextField *address;

-(IBAction) loadPreviousPage: (id) sender;
-(IBAction) loadNextPage: (id) sender;
-(IBAction) loadURLFrom: (id) sender;
@end
```

Im IB verbinden wir das `address`-Outlet mit dem Textfeld. Ich habe auch den Standardwert für das Textfeld im Attributes Inspector entfernt, aber das spielt eigentlich keine Rolle. In Xcode synthetisieren wir `address` in `BrowserController.m` und fügen `awakeFromNib` ein:

CreatingAController/SimpleBrowser6/BrowserController.m

```
-(void)awakeFromNib {
    [self.address setStringValue:@"http://pragprog.com"];
    [self loadURLFrom:self.address];
}
```

8.11 Die Buttons aktivieren und deaktivieren

In Sachen Benutzerfreundlichkeit weist unser Webbrowser immer noch einige Mängel auf. So sind unsere Buttons beispielsweise permanent aktiv, was so aussieht, als könnte der Benutzer sie immer anklicken.

5 Es wird keine Nachricht an Objekte gesendet, die diese Nachricht nicht implementieren, damit es nicht zu Laufzeitfehlern kommt.

Wenn wir das nur aus Sicht eines Objective-C-Programmierers betrachten, ist das auch in Ordnung. Es wird keine Nachricht an Objekte gesendet, die diese Nachricht nicht implementieren, also gibt es keine Laufzeitfehler. Wir können `goBack:` an den Web-View senden, sooft wir wollen: Solange es keine vorige Seite zu laden gibt, wird er es gar nicht erst versuchen.

Doch was die Cocoa-Programmierung so besonders macht, ist die Tatsache, dass wir die Anwendung aus Sicht des Benutzers betrachten müssen. Wenn das Anklicken eines Buttons keinen Sinn ergibt, sollte uns ein visueller Hinweis das wissen lassen. In diesem Abschnitt wollen wir Code entwickeln, der die Buttons aktiviert und deaktiviert.

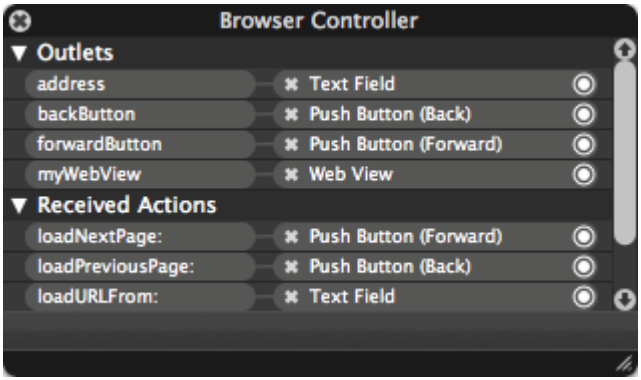
Bevor Sie weiterlesen, sollten Sie versuchen, das selbst zu implementieren, da die Lösung im nachfolgenden Code zu finden ist.

Wir müssen Nachrichten an die beiden Buttons senden, als nehmen wir zwei zusätzliche Outlets in die Header-Datei hinein:

CreatingAController/SimpleBrowser7/BrowserController.h

```
#import <Cocoa/Cocoa.h>
@class WebView;
@interface BrowserController : NSObject {
    WebView *myWebView;
    NSTextField *address;
►   NSButton *backButton;
►   NSButton *forwardButton;
}
@property(assign) IBOutlet WebView *myWebView;
► @property(assign) IBOutlet NSButton *backButton;
► @property(assign) IBOutlet NSButton *forwardButton;
@property(assign) IBOutlet NSTextField *address;

-(IBAction) loadPreviousPage: (id) sender;
-(IBAction) loadNextPage: (id) sender;
-(IBAction) loadURLFrom: (id) sender;
@end
```

Speichern Sie die Header-Datei und wechseln Sie zurück in das Dokumentenfenster des Interface Builder. Wählen Sie den `BrowserController` und öffnen Sie den Connections Inspector. Sie sollten die neuen Outlets `backButton` und `forwardButton` vorfinden. Verknüpfen Sie sie mit den entsprechenden Buttons. Der `BrowserController` sollte nun über vier Outlets und drei empfangene Aktionen verfügen.



Bevor Sie weitermachen, klicken Sie den *Back*-Button an und öffnen den Attributes Inspector (das Icon ganz links im Inspector-Fenster). Bewegen Sie sich nach unten, bis Sie die *Enabled*-Checkbox in der *Control*-Gruppe finden. Deaktivieren Sie die Checkbox. Machen Sie das auch mit dem *Forward*-Button. Speichern Sie ab. Beim Start des Browsers sind nun beide Buttons deaktiviert.

Denken wir mal kurz darüber nach, was mit den Buttons genau geschehen soll. Beim *Back*-Button soll der Button aktiviert oder deaktiviert werden, je nachdem, ob der Web-View eine vorherige Seite laden kann. Der Zustand des *Forward*-Buttons hängt wiederum davon ab, ob sich der Web-View vorwärtsbewegen kann oder nicht. Ein kurzer Blick in die Dokumentation von `NSButton` und `WebView` zeigt, dass man die Methoden `canGoBack` und `canGoForward` nutzen kann, um die Buttons zurückzusetzen:

CreatingAController/SimpleBrowser7/BrowserController.m

```
-(void) resetButtons {
    [self.backButton setEnabled:[self.myWebView canGoBack]];
    [self.forwardButton setEnabled:[self.myWebView canGoForward]];
}
```

Wir müssen `resetButtons` nun in den Aktionsmethoden im `BrowserController` aufrufen.

```
[self resetButtons];
```

Wir deklarieren `resetButtons` nicht in der Header-Datei, weil andere Objekte die Nachricht nicht an unseren Controller senden sollen. Mit anderen Worten ist `resetButtons` nicht Teil der öffentlichen Schnittstelle des `BrowserController`. Um den Compiler zufriedenzustellen, steht `resetButtons` am Anfang der Implementierung, damit die anderen Methoden sie kennen.

Die Reihenfolge ist wichtig

Stellen Sie sich vor, Sie wären der Compiler, der sich durch die Objective-C-Implementierung der Klasse `BeispielKlasse` arbeitet. Während Sie die Methode `foo` durchgehen, stoßen Sie auf eine Referenz für die Methode `bar`, die ebenfalls in `BeispielKlasse` definiert ist.

Es gibt grundsätzlich zwei Möglichkeiten, etwas über `bar` zu erfahren. Zum einen könnte `bar` in der Header-Datei deklariert sein. In diesem Fall wissen Sie und jeder andere, der die Header-Datei für `BeispielKlasse` importiert, über `bar` Bescheid. Zum anderen kann `bar` vor `foo` definiert sein. In diesem Fall nickt der Compiler und sagt: „Ah, über dich habe ich etwas gelesen!"

Wenn Sie nichts über `bar` gehört haben, geben Sie ein Warnung aus. Doch der Programmierer muss sich keine Sorgen machen. Die Warnung besagt, dass ein Objekt des Typs `BeispielKlasse` möglicherweise nicht auf die Methode `bar` reagiert. Da `BeispielKlasse` die Methode `bar` aber tatsächlich implementiert, gibt es (trotz der Warnung während der Kompilierung) zur Laufzeit keine Probleme.

Hier der aktuelle Stand unserer Implementierung:

CreatingAController/SimpleBrowser7/BrowserController.m

```
#import "BrowserController.h"
#import <WebKit/WebKit.h>

@implementation BrowserController

@synthesize myWebView, address, backButton, forwardButton;

-(void) resetButtons {
    [self.backButton setEnabled:[self.myWebView canGoBack]];
    [self.forwardButton setEnabled:[self.myWebView canGoForward]];
}
-(IBAction) loadPreviousPage: (id) sender{
    [self.myWebView goBack:sender];
    [self resetButtons];
}
-(IBAction) loadNextPage: (id) sender{
    [self.myWebView goForward:sender];
    [self resetButtons];
}
-(IBAction) loadURLFrom: (id) sender{
    [self.myWebView takeStringURLFrom:sender];
    [self resetButtons];
```

```
}
-(void)awakeFromNib {
    [self.address setStringValue:@"http://pragprog.com"];
    [self loadURLFrom:self.address];
}
@end
```

Das sieht gut aus. Speichern Sie ab, kompilieren Sie die Anwendung und probieren Sie sie aus.

8.12 Korrekturen nötig

Oha! Die Buttons funktionieren jetzt schlechter als vorher. Vor unseren letzten Änderungen waren die Buttons immer aktiv. Das Problem bestand darin, dass der Benutzer die Buttons anklicken konnte, ohne dass etwas passierte. Nun ist das Gegenteil der Fall: Manchmal sind die Buttons nicht aktiv, obwohl sie es sein sollten. Das liegt an zwei Problemen, die miteinander zu tun haben.

Um das eine Problem zu sehen, starten Sie die Anwendung und geben eine URL ein. Sobald die Seite geladen ist, geben Sie eine weitere URL ein. Der *Back*-Button sollte jetzt eigentlich aktiviert werden, wird er aber nicht. Sobald die zweite Seite geladen wurde, geben Sie eine dritte URL ein. Nun ist der *Back*-Button aktiv und Sie können zurück bis zur ersten Seite navigieren. Das Problem besteht darin, dass es eine Weile dauert, bis die URL geladen ist, während Sie den Status des Buttons direkt nach dem Lade-Request gesetzt haben. Es wäre besser, `canGoBack:` und `canGoForward:` aufzurufen, nachdem das Laden der URL angestoßen wurde.

Beenden Sie die Anwendung und starten Sie sie erneut. Geben Sie eine URL ein. Sobald die Seite geladen ist, klicken Sie einen Link an. Nach dem Laden folgen Sie einem weiteren Link. Solange Sie den Links im Web-View folgen, werden die Buttons nie aktiviert. Die History wird gepflegt, doch es gibt kein Callback für `resetButtons`. Sie merken das, wenn Sie eine URL eingeben. Sobald die Seite geladen ist, können Sie sich über die Buttons in der History vor- und zurückbewegen.

Alle Methoden, die Sie bisher gesehen haben, werden sofort ausgeführt. Was wir benötigen, ist eine Möglichkeit, um das Senden einer Nachricht aufzuschieben: „Frag mich nicht, ob ich vor- oder zurückgehen kann, bis ich die angeforderte Seite geladen habe!“ Glücklicherweise ist dieser Benachrichtigungsmechanismus in Form von Delegates in Cocoa integriert. Wir werden uns im nächsten Kapitel ansehen, wie sie funktionieren.

Kapitel 9

Anpassungen mit Delegates

Es passiert etwas.

Es passiert sogar eine ganze Menge, wenn der Benutzer im SimpleBrowser URLs eingibt und Buttons oder Links anklickt. Der grundlegendste Ereignistyp ist „Ziel/Aktion“ (target-action). Der Benutzer klickt einen Button an, und eine Aktion wird an ein Ziel gesendet. Sie haben zwei Möglichkeiten kennengelernt, um mit solchen Ereignissen umzugehen: Sie können den Interface Builder benutzen, um das Objekt, das die Nachricht sendet, direkt mit dem Zielobjekt zu verknüpfen, das die Aktion ausführt, oder Sie können einen Controller entwickeln.

Doch sobald Sie eine URL eingeben oder den *Forward*- oder *Back*-Button benutzen, rauschen zusätzliche Ereignisse und Nachrichten an uns vorbei, von denen wir gar nichts mitbekommen. Beispielsweise wird Ihnen aufgefallen sein, dass, wenn Sie eine Seite in Safari laden, der Titel der neuen Seite über der Symbolleiste erscheint, bevor die Seite geladen ist. Gleichzeitig können Sie den Ladefortschritt über den blauen Balken verfolgen, der im Textfeld mit der URL erscheint. Es werden also Nachrichten gesendet, und Safari kann sie abfangen und verarbeiten. Was ist mit uns?

Die ganze Zeit über schwirren jede Menge Nachrichten an uns vorbei. In diesem Kapitel werden Sie lernen, sie abzufangen und auf sie zu reagieren. Wir werden den Titel der zu ladenden Webseite ausgeben und die Buttons und die URL korrigieren. Delegates ermöglichen es uns, das Verhalten einer Klasse anzupassen, ohne eine Subklasse entwickeln zu müssen.

9.1 Delegates verstehen

Bevor wir Delegates auf unser Browserbeispiel anwenden, wollen wir die `NSWindow`-Klasse nutzen, um zu untersuchen, wie Delegates funktionieren.

Stellen Sie sich vor, dass unser `NSWindow`-Objekt in einer Cocoa-Gameshow auftritt. Wenn Sie sich die Dokumentation für `NSWindow` anschauen, sehen Sie mehr als 200 Methoden, die als Tasks aufgeführt sind, die das Fenster durchführen könnte.

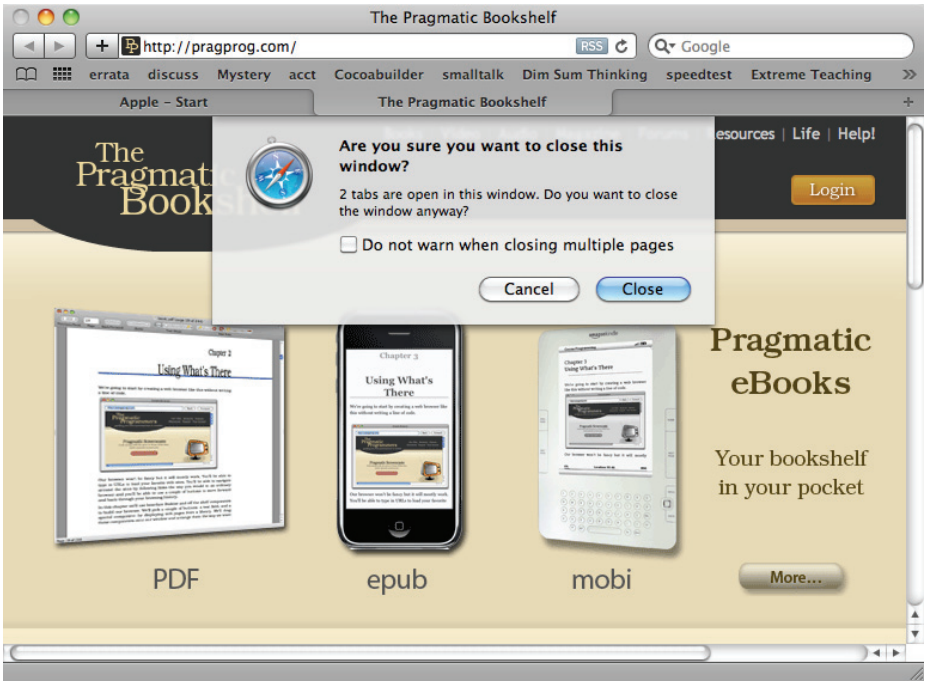
Unser Window-Objekt ist recht zuversichtlich, auf die meisten empfangenen Nachrichten richtig reagieren zu können. Zum Beispiel weiß es, wie es auf eine `setShowsToolbarButton:`-Nachricht reagieren soll: Besitzt das Fenster eine Symbolleiste, wird sie bei der Übergabe von `YES` ausgegeben, während die Übergabe von `NO` sie versteckt. Besitzt unser Fenster keine Symbolleiste, dann passiert nichts weiter, wenn die Methode aufgerufen wird.

Andererseits gibt es einige Tasks, bei denen unser Fenster nicht weiß, wie es reagieren soll. Was passiert beispielsweise, wenn unser Fenster die folgende Frage gestellt bekommt?

Die Antwort ist nicht so einfach, wie Sie vielleicht glauben. Wenn jemand den roten Button anklickt, wird das Fenster üblicherweise geschlossen, die Anwendung aber weiterhin ausgeführt. Das entspricht dem Standardverhalten, ist aber nicht die einzige Möglichkeit. Wenn Sie zum Beispiel die Systemeinstellungen öffnen und den roten Button anklicken, wird das Fenster geschlossen und die Anwendung beendet.

Eine weitere Verhaltensweise sehen Sie, wenn Sie bei Safari eine Webseite öffnen und dann einen neuen Tab (bei aktivierten Tabs wählen Sie `Ablage > Neuer Tab` oder ⌘T). Öffnen Sie eine Webseite im neuen Tab und klicken Sie dann den roten Button an.

Obwohl Sie in derselben Anwendung denselben roten Button angeklickt haben, wird das Fenster nicht automatisch geschlossen, wenn mehrere Tabs geöffnet sind. Stattdessen werden Sie gefragt, ob Sie „dieses Fenster wirklich schließen wollen“.



Diese Varianten stellen für unser Fenster ein Dilemma dar. Was soll es antworten, wenn der Quizmaster fragt: „Was würden Sie tun, wenn ich ihren roten Button anklicke?“

Das Fenster möchte Antworten „das kommt drauf an“, aber das ist weder in einer Gameshow noch in einer laufenden Cocoa-Anwendung eine akzeptable Antwort. Also entscheidet sich das Fenster für den Telefonjoker und ruft einen Freund an.

Es folgen die Regeln dieses Spiels. Das Fenster identifiziert vorab alle Nachrichten, bei denen es Hilfe brauchen könnte, und kennzeichnet sie als Delegate-Methoden.[1] Nachfolgend sehen Sie alle Delegate-Methoden für die `NSWindow`-Klasse:

1 In diesem Kapitel nutzen wir von Apple entwickelte Protokolle und Delegates. Einen eigenen Delegate und ein eigenes Protokoll entwickeln wir in Kapitel 12, *Protokolle für Delegates entwickeln*, auf Seite 203.

```
window:shouldDragDocumentWithEvent:from:withPasteboard:
window:shouldPopUpDocumentPathMenu:
window:willPositionSheet:usingRect:
windowDidBecomeKey:
windowDidBecomeMain:
windowDidChangeScreen:
windowDidChangeScreenProfile:
windowDidDeminiaturize:
windowDidEndSheet:
windowDidExpose:
windowDidMiniaturize:
windowDidMove:
windowDidResignKey:
windowDidResignMain:
windowDidResize:
windowDidUpdate:
windowShouldClose:
windowShouldZoom:toFrame:
windowWillBeginSheet:
windowWillClose:
windowWillMiniaturize:
windowWillMove:
windowWillResize:toSize:
windowWillReturnFieldEditor:toObject:
windowWillReturnUndoManager:
windowWillUseStandardFrame:defaultFrame:
```

Die Delegate-Methoden gibt es in drei Varianten: „Etwas `wird` passieren“, „etwas `ist` passiert“ und „etwas `sollte` passieren“. Sie implementieren die `wird`-Version, um das Verhalten zu ändern, bevor die Aktion angestoßen wird, und die `ist`-Variante als Reaktion auf eine bereits ausgeführte Aktion. Die `sollte`-Version gibt ein `BOOL`eschen Wert zurück, der Ihnen erlaubt, eine Aktion abzubrechen, wenn Sie glauben, dass sie nicht durchgeführt werden sollte.

Genau wie bei einer Gameshow, bei der der Teilnehmer „einen Freund anrufen“ kann, muss jedes Fensterobjekt einen Freund angeben, der „angerufen“ werden kann, wenn eine dieser Methoden aufgerufen wird. Dieser Freund ist der Delegate.

Wir erwarten nicht, dass ein Freund in der Lage ist, auf all diese Nachrichten zu reagieren, aber unser Fenster muss sich im Vorfeld für genau einen Delegate entscheiden. Leider kann das Fenster nicht sagen, „ich weiß, wer diese Frage beantworten kann“, und zur Laufzeit entscheiden, wer „angerufen“ wird. Der Delegate wird gewählt, bevor die Frage gestellt wird.

Bevor das Spiel beginnt, wird der Delegate interviewt. Auf diese Weise wissen wir, welche Nachrichten der Delegate verarbeiten kann und welche nicht. Nach Cocoa-Lesart kann der Delegate keine, einige oder alle Dele-

gate-Methoden für ein Objekt implementieren. Hat ein Delegate eine Methode nicht implementiert, wird er für die Bearbeitung dieser Nachricht nicht aufgerufen. Das Fenster muss dann eine eigene Antwort liefern.

Diese Regeln klingen kompliziert, doch ein kurzes Beispiel sollte den Sachverhalt klären.

9.2 Das Standardverhalten eines Fensters

Legen Sie in Xcode ein neues Projekt an. Es wird eine `Mac OS X > Application > Cocoa Application` namens *WindowDressing*. Wir werden einen Delegate verwenden, um das Standardverhalten des grünen und roten Buttons zu ändern.

Klicken Sie *Build & Run* an. Wenn Sie den grünen Button anklicken, wird das Fenster vergrößert und füllt den Großteil des Bildschirms aus. Klicken Sie ihn erneut an, kehrt das Fenster zu seiner ursprünglichen Größe zurück. Klicken Sie den roten Button an, und das Fenster wird geschlossen, die Anwendung aber nicht beendet. Sobald Sie den roten Button anklicken, können Sie nicht mehr viel machen. Sie können kein neues Fenster öffnen, es bleibt Ihnen also nichts anderes übrig, als die Anwendung zu beenden.

Sie haben jetzt das Standardverhalten für das Anklicken des grünen und roten Buttons gesehen. Dieses Verhalten stellt das Fenster zur Verfügung, wenn es keinen Freund anrufen darf. Im nächsten Abschnitt stellen wir mithilfe eines Delegate ein anderes Verhalten bereit.

9.3 Ein roter Hintergrund

Lassen Sie uns das Verhalten so ändern, dass der Hintergrund des Fensters rot eingefärbt wird, wenn der Benutzer den roten Button anklickt. Dazu sind die folgenden Schritte notwendig:

1. Wir müssen eine neue Klasse entwickeln, die wir instanziieren können, um den Delegate für das Fenster zu erzeugen.
2. Wir müssen dieses Objekt als Delegate für dieses Fenster festlegen.
3. Wir müssen die Delegate-Methode ermitteln, die es zu implementieren gilt.
4. Zum Schluss müssen wir diese Methode implementieren.

Im zweiten und dritten Schritt machen die Leute häufig kleine, schwer zu erkennende Fehler. Man vergisst leicht, das Fenster im Interface Builder mit seinem Delegate zu verknüpfen. Außerdem muss man die *.nib*-Datei sichern, nachdem eine Änderung vorgenommen wurde. Wenn Sie Ihr Projekt in Xcode entwickeln, werden Sie gefragt, ob veränderte Quelldateien gesichert werden sollen, aber im Interface Builder können ebenfalls ungesicherte Änderungen vorliegen.

Ich garantiere Ihnen, dass Sie an irgendeiner Stelle darüber stolpern werden. Wenn Sie zwischen Xcode und Interface Builder hin- und herwechseln und das Projekt sich nicht wie erwartet verhält, werfen Sie einen Blick in das Dokumentenfenster des Nib. Wenn Sie einen dunklen Punkt in der Mitte des roten Schließen-Buttons sehen, wurde die Datei geändert und muss gesichert werden.

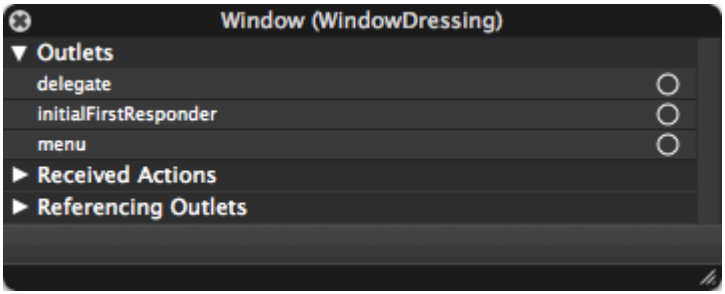
Gibt es keine ungesicherten Änderungen, sollte der rote Schließen-Button so sauber aussehen wie der gelbe Minimieren- und der grüne Maximieren-Button. Dieser dezente Hinweis zieht sich durch alle Mac OS X-Anwendungen, und es ist praktisch, auf ihn zu achten.

Sie müssen auch sorgfältig darauf achten, den Namen der Delegate-Methode genau so zu schreiben, wie er in der `NSWindow`-Dokumentation steht. Als wir mit Buttons gearbeitet haben, konnten wir jedes Objekt als Ziel angeben und den Namen der Methode frei wählen. Bei Delegates wird der Name der Methode für Sie gewählt.

In Xcode erzeugen Sie eine neue Klassendatei über `File` > `New File...` oder ⌘N. Wählen Sie die Vorlage `Mac OS X` > `Cocoa Class` > `Objective-C class` und nennen Sie sie `WindowHelper`.

Genau wie im vorigen Kapitel erzeugen wir nun eine Repräsentation dieser Klasse im Interface Builder. Klicken Sie `MainMenu.xib` doppelt an, um sie im Interface Builder zu öffnen. Suchen Sie `WindowHelper` unter dem *Classes*-Reiter in der Library heraus und ziehen Sie sie in das Dokumentenfenster.[2] Ein Control-Klick auf das Window-Objekt – nicht auf das `WindowHelper`-Objekt – innerhalb des Dokumentenfensters öffnet das Verbindungsfenster.

2 Die Suche geht wesentlich schneller, wenn Sie *Win* in das Suchfeld am unteren Rand des Library-Fensters eingeben.



Ziehen Sie den Kreis neben dem `delegate`-Outlet über das `WindowHelper`-Objekt innerhalb des Dokumentenfensters. Damit wird der Window-Delegate als Instanz von `WindowHelper` festgelegt. Sichern Sie Ihre Arbeit und beenden Sie den Interface Builder.

Unter Xcode müssen Sie eigentlich keine Änderungen an der `WindowHelper`-Header-Datei vornehmen. Sie haben die Verbindung zum Delegate im Interface Builder hergestellt. Als Entwickler kennen Sie die Signatur der Methoden, die Sie implementieren können. Zur Laufzeit kennt das System die Nachrichten, die es senden kann. Sie können aber die Absicht von `WindowHelper` deutlich machen, indem Sie angeben, dass er das `NSWindowDelegate`-Protokoll implementiert.

Ein Protokoll deklariert eine Sammlung von Methoden, die eine Klasse implementieren kann (oder auch nicht). Sie deklarieren ein oder mehrere Protokolle zu Beginn der Header-Datei zwischen spitzen Klammern:

Delegates/WindowDressing2/WindowHelper.h

```
#import <Cocoa/Cocoa.h> ^

► @interface WindowHelper : NSObject <NSWindowDelegate> {
}
@end
```

Wenn Sie sich die Dokumentation für das `NSWindowDelegate`-Protokoll ansehen, erkennen Sie, dass alle Methoden als *optional* aufgeführt sind. Mit anderen Worten steht es Ihnen frei, nur diejenigen zu implementieren, die Sie benötigen.

Es gibt nur zwei Delegate-Methoden, die etwas mit dem Schließen eines Fensters zu tun haben: `windowShouldClose:` und `windowWillClose:`. Die Methode `windowShouldClose:` wird aufgerufen, wenn der Benutzer den roten Button anklickt, und gibt Ihnen die Möglichkeit, das Fenster doch nicht zu schließen. Die Methode `windowWillClose:` wird unmittelbar aufgerufen, bevor das Fenster geschlossen wird.

Wir werden die Methode `windowShouldClose:` verwenden. Weil Schreibfehler sich so leicht einschleichen und man sie nur schwer erkennen und debuggen kann, wechsle ich häufig in die Dokumentation und benutze Ausschneiden und Ersetzen, um die Methodensignatur direkt zu übernehmen.

Die `windowShouldClose:`-Methode muss zweierlei erledigen. Sie soll die Hintergrundfarbe des Fensters auf rot setzen. Diesmal ist das Fenster der `sender`, wir können also den `sender` anweisen, seine Hintergrundfarbe auf rot zu setzen.

Die Methode muss außerdem `NO` zurückgeben, damit das Fenster nicht geschlossen wird. Wenn Sie `YES` zurückgeben, wird das Fenster geschlossen. Hier sehen Sie `WindowHelper.m`:

Delegates/WindowDressing2/WindowHelper.m

```
#import "WindowHelper.h"

@implementation WindowHelper
-(BOOL)windowShouldClose:(id)sender {
    [sender setBackgroundColor:[NSColor redColor]];
    return NO;
}

@end
```

Klicken Sie auf *Build & Run*, und das Fenster wird rot, sobald Sie den roten Button anklicken. Genauer gesagt wird, wenn Sie den roten Button anklicken, die Nachricht `windowShouldClose:` an das Objekt gesendet, das Sie als Delegate für das Window-Objekt festgelegt haben. Beenden Sie WindowDressing mit ⌘Q oder klicken Sie in Xcode auf *Stop*.

9.4 Übung: Grüner Hintergrund

Finden Sie die Methode, die aufgerufen wird, wenn der grüne Button angeklickt wird, um das Fenster zu maximieren. Implementieren Sie die Methode so, dass das Fenster seine Größe nicht ändert, sondern die Hintergrundfarbe auf grün setzt.

9.5 Lösung: Grüner Hintergrund

Sie müssen nur die Methode `windowShouldZoom:toFrame:` implementieren. Der Funktionsrumpf sollte im Wesentlichen mit `windowShouldClose:` übereinstimmen.

Delegates/WindowDressing3/WindowHelper.m

```
-(BOOL)windowShouldZoom:(NSWindow *)window toFrame:(NSRect)newFrame {
    [window setBackgroundColor:[NSColor greenColor]];
    return NO;
}
```

Klicken Sie auf *Build & Run*. Die Hintergrundfarbe des Fensters sollte nun zu rot bzw. grün wechseln, wenn Sie den entsprechenden Button anklicken.

9.6 Application-Delegate

Tatsächlich nutzen wir Delegates schon seit unserem ersten Projekt. Die in Xcode 3.2 eingeführte Cocoa Application-Schablone enthält einen Application-Delegate. Nachdem Sie die grundlegende Idee im Zusammenhang mit einem Fenster verstanden haben, sollten Sie erkennen können, wie es bei Anwendungen eingesetzt wird.

Beim Start einer Anwendung passiert vielerlei, und Apple kümmert sich um das meiste davon. Aber Ihre Anwendung kann auch so spezielle Dinge umfassen, dass Apple sich unmöglich darum kümmern kann. Wenn es etwas Bestimmtes gibt, das erledigt werden soll, sobald die Anwendung geladen wurde, dann implementieren sie die `applicationDidFinishLaunching:`-Methode des Application-Delegate.

Beachten Sie, dass die Header-Datei des App-Delegate die Protokolldeklaration `NSApplicationDelegate` umfasst und dass das `delegate`-Outlet im Interface Builder aus der Anwendung mit dem App-Delegate verknüpft ist.

Wenn Sie sich die Dokumentation zu `NSApplicationDelegate` ansehen, finden Sie Methoden, die ein spezialisiertes Verhalten Ihrer Anwendung an Dutzenden von Schlüsselstellen im Leben Ihrer Anwendung erlauben. Es gibt Delegate-Methoden für das Starten und Beenden der Anwendung, zum Verstecken der Anwendung, zur Verwaltung ihrer Fenster und so weiter.

Delegates sind ein wichtiges Entwurfsmuster für Cocoa-Anwendungen. Sie finden sie hauptsächlich in modernen Cocoa-Desktop-APIs wie WebKit. Delegates werden auch durchweg in den iPhone-APIs genutzt.

9.7 Delegates für ihren Web-View

Der Rest dieses Kapitels wendet unser neu erworbenes Wissen über Delegates auf unser *SimpleBrowser*-Beispiel an. Schließen Sie das *WindowDressing*-Projekt und öffnen Sie wieder das *SimpleBrowser*-Projekt.

Wenn Sie das Verhalten einer Klasse ändern wollen, erzeugen Sie bei der objektorientierten Programmierung häufig eine Subklasse und überschreiben eine oder mehrere Methoden. Die Delegation erlaubt es uns, Vererbungsketten zu vermeiden, wenn wir eine Basisklasse ein wenig modifizieren müssen. Mit der Delegation identifizieren wir die Methoden, die am häufigsten geändert werden müssen, kennzeichnen Sie als Delegate-Methoden und stellen ein Standardverhalten zur Verfügung; dieses wird angewendet, solange wir keinen Delegate bereitstellen und keine Methode mit einem speziellen Verhalten implementieren.

Ein Objekt vom Typ `NSWindow` kann ein einzelnes Delegate-Objekt spezifizieren. Nach dieser Variante funktionieren Delegates bei Cocoa üblicherweise. Manchmal gibt es aber auch Klassen mit mehreren Delegates. Die Entwickler der `WebView`-Klasse haben sich zum Beispiel das ganze Verhalten angesehen, das Sie möglicherweise anpassen wollen, und in vier Delegates eingruppiert.

Diese vier Delegates sind in der Übersicht der `WebView`-Klassenreferenz aufgeführt:

- `WebFrameLoadDelegate`
- `WebPolicyDelegate`
- `WebResourceLoadDelegate`
- `WebUIDelegate`

Sie können sich den `WebView` in unserer Cocoa-Gameshow vorstellen, aber diesmal kann er einen von vier Freunden anrufen, je nachdem, aus welcher Kategorie die Frage stammt.

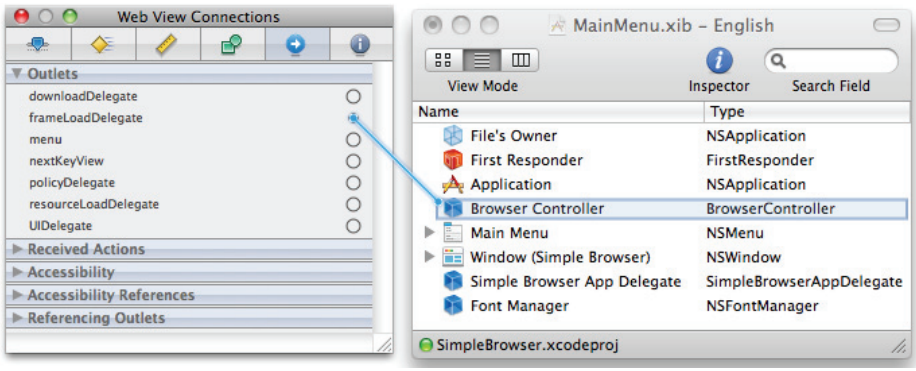
Diese vier Delegate-Gruppen sind Protokolle und keine Klassen. Sie geben die Methodensignaturen an, die eine Klasse nutzen kann, die das Protokoll implementiert. Auf diese Weise wird sichergestellt, dass die passende Methode aufgerufen wird, wenn eine Nachricht an das Delegate-Objekt gesendet wird. Das bedeutet auch, dass der Compiler Ihnen Bescheid sagen kann, wenn es ein Problem gibt.

Sehen Sie sich den „Tasks“-Abschnitt der `WebView`-Dokumentation unter „Getting and Setting Delegates“ an: Sie sehen zehn Methoden zur Zuweisung und zur Arbeit mit Delegates. Sie können Delegates über Outlets im Interface Builder oder programmtechnisch über eine Methode wie `setUIDelegate:` festlegen.

9.8 Den Titel des Fensters setzen

Als Nächstes wollen wir die Titelleiste des Browserfensters mit dem Titel der geladenen Seite füllen. Wir benutzen dazu die Methode `webView:didReceiveTitle:forFrame:` im `WebFrameLoadDelegate`.

Welches Objekt soll unser Delegate sein? Wir können entweder eine neue Klasse entwickeln und instanziieren oder die Instanz einer bereits existierenden Klasse nutzen. In diesem Fall ist das `BrowserController`-Objekt die beste Wahl, weil es Verbindungen zu den GUI-Elementen besitzt, auf die wir zugreifen müssen. Verbinden Sie den Delegate im Interface Builder so:



Wählen Sie den Web-View und öffnen Sie den Connections Inspector. Sie sehen die vier Delegate-Methoden zusammen mit den anderen Outlets. Verbinden Sie `frameLoadDelegate` mit dem `BrowserController`. Speichern Sie und beenden die Arbeit in Xcode.

Die Methode `webView:didReceiveTitle:forFrame:` wird aufgerufen, sobald der Titel verfügbar ist. Sie können erkennen, dass die empfangene Nachricht auch den Titel als Parameter enthält. Wir nutzen sie wie folgt:

- Wir fügen *nichts weiter* in die Header-Datei ein. Alle 15 Methoden im `WebFrameLoadDelegate`-Protokoll stehen uns nun im `BrowserController` zur Verfügung. Die Dokumentation sagt uns, dass `WebFrameLoadDelegate` ein informelles Protokoll ist; wir deklarieren es also nicht in der `BrowserController`-Header-Datei.

- Wir müssen `webView:didReceiveTitle:forFrame:` in `BrowserController.m` implementieren, indem wir den Rumpf der folgenden Methode auffüllen:

```
-(void)webView:(WebView *)sender didReceiveTitle:(NSString *)title
                                          forFrame:(WebFrame *)frame{
}
```

  Wir haben die Signatur der Methode aus der Dokumentation des `WebFrameLoadDelegate`-Protokolls kopiert.

- Wir warten. Die `webView:didReceiveTitle:forFrame:`-Methode wird aufgerufen, sobald der Titel der zu ladenden URL verfügbar ist.

Wenn die Methode aufgerufen wird, wollen wir das `title`-Attribut des Anwendungsfensters auf den an uns zurückgegebenen Titel setzen. Ein Blick auf die Methodensignatur zeigt, dass ein Handle auf den verwendeten Web-View und der Titel der neuen Seite übergeben werden. Sie können den Web-View nutzen, um einen Zeiger auf das Fenster zu ermitteln, das den Web-View enthält:

```
[sender window];
```

Unglücklicherweise finden Sie die benötigte Methode nicht in der `WebView`-Dokumentation. Vielmehr müssen Sie sich in der Superklasse `NSView` umsehen, in der Sie die `window`-Methode finden. Die `window`-Methode wird in `NSView` auch als Methode hervorgehoben, die „ein `NSWindow`-Objekt zurückgibt, das das `NSView`-Objekt enthält“.

Sobald Sie das Fenster kennen, können Sie seinen Titel mit dem `title` festlegen, der beim Aufruf der Delegate-Methode übergeben wurde. Fügen Sie die folgende Methode in `BrowserController.m` ein:

Delegates/SimpleBrowser8/BrowserController.m

```
 -(void)webView:(WebView *)sender
didReceiveTitle:(NSString *)title
       forFrame:(WebFrame *)frame {
    [[sender window] setTitle:title];
}
```

Speichern Sie ab. Klicken Sie auf *Build & Run*, und egal, wie Sie zu einer Website navigieren – der Titel erscheint im Browser, sobald er zur Verfügung steht. Sie mussten nur die richtige Methode finden, die Outlets erzeugen und konfigurieren und eine Methode mit einer einzige Zeile Code entwickeln:

```
[[sender window] setTitle:title];
```

Wieder haben wir die Methode `webView:didReceiveTitle:forFrame:` *nicht* im `BrowserController`-Header eingetragen. Die Methode ist nicht Teil der öffentlichen Schnittstelle von `BrowserController`. Das einzige Objekt, das wissen muss, dass `BrowserController` diese Methode implementiert, ist das delegierende Objekt `myWebView`. Die Nachricht wird nur gesendet, wenn der Delegate zugewiesen und die Methode implementiert wird.

9.9 Übung: URL aktualisieren und Buttons setzen

Nachdem Sie erfahren haben, wie man den Titel der Webseite als Titel des Fensters festlegt, können Sie die Buttons zurücksetzen und die URL aktualisieren, sobald die Seite geladen ist. Klickt ein Benutzer einen Link auf einer Webseite an, dann ist es bisher so, dass die Seite zwar geladen, die URL im Textfeld aber nicht aktualisiert wird (was man so ziemlich von jedem anderen Browser kennt).

Um die Buttons zurückzusetzen und die URL nach dem Laden der Seite zu aktualisieren, müssen Sie eine Methode im `WebFrameLoadDelegate`-Protokoll finden, die aufgerufen wird, wenn der Frame vollständig geladen wurde. Die Delegate-Methode muss dann zwei Dinge tun: (a) den Stringwert des Textfelds auf den URL des Hauptframes der Seite setzen und (b) `resetButtons` aufrufen.

Tipp 1

Wenn Sie in der Klassenreferenz zu `NSTextField` nach einer Methode suchen, die den Stringwert eines Textfeldes setzt, werden Sie nicht fündig werden. Denken Sie daran, dass vererbte Methoden nicht in der Cocoa-Dokumentation auftauchen. Sie müssen sich also die Klassenreferenz der `NSTextField`-Superklasse `NSControl` ansehen. Sie finden `setStringValue:` in „Tasks“ unter „Setting the Control's Value.“

Tipp 2

`WebView` besitzt eine Vielzahl von Methoden. Die zur Abfrage der URL kann man sehr leicht übersehen. Suchen Sie im Abschnitt „Tasks“ unter der Überschrift „Getting and Setting Frame Contents“.

Codevervollständigung

Sie können die Code Sense-Einstellungen von Xcode unter *Preferences* anpassen. Sie können festlegen, wie schnell die Codevervollständigung erscheint. Statt sich bei der Suche nach einer Methode durch die gesamte Hierarchie hangeln zu müssen, können Sie sich mit `Edit > Completion List` die verfügbaren Möglichkeiten ansehen. Im Beispiel aus der Übung würde nach der Eingabe von `[inputField` in der Liste `setStringValue:` auftauchen.

9.10 Lösung: URL aktualisieren und Buttons setzen

Implementieren Sie die the `webView:didFinishLoadForFrame`-Methode so:

Delegates/SimpleBrowser9/BrowserController.m

```
-(void)webView:(WebView *)sender didFinishLoadForFrame:(WebFrame *)frame {
    [self.address setStringValue:[sender mainFrameURL]];
    [self resetButtons];
}
```

Klicken Sie auf *Build & Run*. Der Code sollte einwandfrei funktionieren. Der Titel des Fensters wird zum richtigen Zeitpunkt gesetzt, die URL ändert sich, während Sie sich mit den Buttons vor- und zurückbewegen, und die Buttons werden korrekt (de)aktiviert. Sie werden jetzt zurecht stolz auch sich sein, doch bevor Sie mit dem nächsten Kapitel weitermachen, müssen Sie noch ein wenig aufräumen.

Der Code ist ein ziemliches Durcheinander. Es gibt viel Redundanz, und wahrscheinlich können wir einige Outlets weglassen. An unserem Vorgehen ist aber nichts verkehrt. Zuerst bringen wir den Code zum Laufen, und dann räumen wir auf.

9.11 Aufräumen

Es gibt einige überflüssige Dinge, um die wir uns kümmern müssen. Sehen Sie sich jeweils die letzte Zeile der folgenden Methoden an:

Delegates/SimpleBrowser9/BrowserController.m

```
-(IBAction) loadPreviousPage: (id) sender{
    [self.myWebView goBack:sender];
►   [self resetButtons];
}
```

```
-(IBAction) loadNextPage: (id) sender{
    [self.myWebView goForward:sender];
►   [self resetButtons];
}

-(IBAction) loadURLFrom: (id) sender{
    [self.myWebView takeStringURLFrom:sender];
►   [self resetButtons];
}
```

In keiner dieser Methoden müssen wir `resetButtons` aufrufen, weil das schon geschieht, sobald der Frame geladen wurde. Entfernen Sie diese drei Zeilen und führen Sie die Anwendung erneut aus. Sie werden sehen, dass sie immer noch korrekt funktioniert.

Nachdem wir `[self resetButtons];` aus `loadPreviousPage:`, `loadNextPage:` und `loadURLFrom:` entfernt haben, machen die Methoden nicht mehr allzu viel. Tatsächlich gibt es eigentlich keinen Grund mehr, den Controller für diese Aktionen zu verwenden.

Um die Sachlage zu verdeutlichen, möchte ich diese Methoden kurz unangetastet lassen und in den Interface Builder wechseln. Klicken Sie den `BrowserController` im Dokumentenfenster an und wählen Sie den Connections Inspector. Trennen Sie die drei empfangenen Aktionen durch einen Klick auf das entsprechende X. Wählen Sie nun den WebView und nutzen Sie den Connections Inspector, um diese drei Aktionen erneut zu verknüpfen. Ziehen Sie von `goBack:` zum *Back*-Button, von `goForward:` zum *Forward*-Button und von `takeStringURLFrom:` zum Textfeld. Sichern Sie Ihre Arbeit im Interface Builder und klicken Sie dann *Build & Run* in Xcode an – die Anwendung funktioniert ausgezeichnet.

Das sollte Sie etwas beunruhigen.

Sie haben den Kontrollfluss völlig aus dem IB heraus umgeleitet. Sie haben in Xcode drei Methoden, die nicht mehr aufgerufen werden. Das ist etwas, woran Sie denken müssen, wenn Sie mit Cocoa-Programmen arbeiten. Sie können das große Ganze nicht erkennen, wenn Sie sich nur den Code ansehen. Sie müssen sich auch die Verbindungen ansehen, die Sie an anderer Stelle hergestellt haben.[3]

3 Wenn Sie testgesteuert entwickeln, sollten Sie sich Chris Hansons Blog-Postings zum Unit-Testing von Cocoa-Benutzerschnittstellen unter *http://eschatologist.net/blog/?p=205* sowie seine einführenden Artikel zum Unit-Testing von Cocoa-Code unter *http://eschatologist.net/blog/?p=24* und *http://chanson.livejournal.com/119303.html* ansehen.

Doch Sie sollten es anderen leichter machen, die auf ein Projekt zurückkommen und herausfinden müssen, was vor sich geht. (Und auch sich selbst sollten Sie es leichter machen.) Da diese drei Methoden nicht mehr aufgerufen werden, sollten Sie sie sowohl aus dem Header als auch aus der Implementierung entfernen.

Sobald sie entfernt sind, müssen Sie die Methode `awakeFromNib` so refaktorieren, dass die URL an den Web-View übergeben wird, ohne durch eine der gerade gelöschten Methoden zu laufen. Hier der aktuelle Stand unserer Implementierungsdatei:

Delegates/SimpleBrowser10/BrowserController.m

```
#import "BrowserController.h"
#import <WebKit/WebKit.h>

@implementation BrowserController

@synthesize myWebView, address, backButton, forwardButton;

-(void) resetButtons {
    [self.backButton setEnabled:[self.myWebView canGoBack]];
    [self.forwardButton setEnabled:[self.myWebView canGoForward]];
}

-(void)awakeFromNib {
    [self.address setStringValue:@"http://pragprog.com"];
    [self.myWebView takeStringURLFrom:self.address];
}

-(void)webView:(WebView *)sender
didReceiveTitle:(NSString *)title
        forFrame:(WebFrame *)frame {
    [[sender window] setTitle:title];
}

-(void)webView:(WebView *)sender
            didFinishLoadForFrame:(WebFrame *)frame {
    [self.address setStringValue:[sender mainFrameURL]];
        [self resetButtons];
}

@end
```

Hier die aktuelle Header-Datei:

Delegates/SimpleBrowser10/BrowserController.h

```
#import <Cocoa/Cocoa.h>
@class WebView;
@interface BrowserController : NSObject {
    NSTextField *address;
    NSButton *backButton;
```

```
    NSButton *forwardButton;
    WebView *myWebView;
}
@property(assign) IBOutlet NSButton *backButton;
@property(assign) IBOutlet NSButton *forwardButton;
@property(assign) IBOutlet NSTextField *address;
@property(assign) IBOutlet WebView *myWebView;
@end
```

Beachten Sie, wie wenig Code wir benötigen, wenn wir *mit* den vorhandenen Apple-Frameworks arbeiten. Wenn Ihr Code lang und kompliziert wird, sollten Sie kurz innehalten und überlegen, ob es nicht eine einfachere Lösung für das gibt, was Sie erreichen wollen. Ihr Ziel ist einfacher Code, der klar und verständlich ist. Cocoa-Programmierer mögen es nicht, wenn Code zwar kurz und raffiniert, aber unverständlich ist.

9.12 Übung: Eine Fortschrittsanzeige einbinden

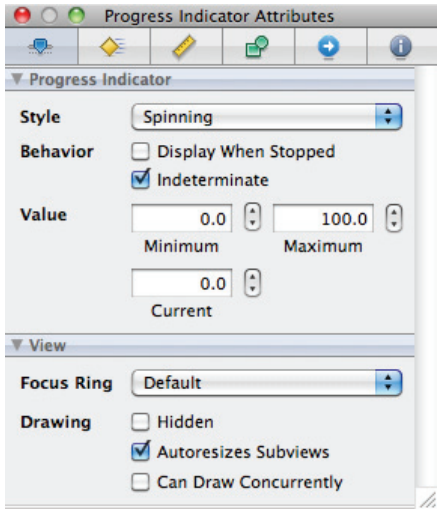
Ein Großteil dessen was wir hier tun, soll Sie an den ständigen Wechsel zwischen Xcode und Interface Builder gewöhnen. Lassen Sie uns also noch einen Schritt weiter gehen – zumindest auf dem Desktop.

Fügen Sie eine kleine runde Fortschrittsanzeige rechts oben in Ihrem Browser ein. Diese sollte zu Beginn nicht zu sehen sein. Wenn der Benutzer eine neue URL eingibt oder sich auf eine andere Seite bewegt, soll die Forschrittsanzeige erscheinen und sich drehen. Wir wollen dem Benutzer zeigen, dass etwas passiert. Sobald die Seite geladen ist, soll sich die Fortschrittsanzeige nicht mehr drehen und wieder verschwinden.

Denken Sie über die nötigen Schritte nach. Sie müssen eine Fortschrittsanzeige mithilfe des IB platzieren. Sie müssen ein Outlet für diese Anzeige in der `BrowserController`-Header-Datei einfügen und im IB verknüpfen. Sie müssen die gerade angelegte Eigenschaft synthetisieren und einsetzen, wenn das Laden einer Seite beginnt und endet.

9.13 Lösung: Eine Fortschrittsanzeige einbinden

Sie können sich die Lösung im Codedownload ansehen, um die *.nib*- und die Header-Datei zu untersuchen. Hier sehen Sie die Einstellungen des Attributes Inspector für die Fortschrittsanzeige:



Unter *Behavior* habe ich „Display When Stopped“ deaktiviert, und unter *Drawing* „Hidden“.

Nun zur endgültigen Version der Implementierungsdatei. Wir müssen einige Zeilen Code in die Methode `webView:didFinishLoadForFrame:` einfügen, um die Animation der Fortschrittsanzeige anzuhalten und um sie zu verstecken. Andersherum müssen wir die Fortschrittsanzeige in der `webView:didStartProvisional-LoadForFrame:`-Methode sichtbar machen und die Animation starten.

Delegates/SimpleBrowser11/BrowserController.m

```
#import "BrowserController.h"
#import <WebKit/WebKit.h>

@implementation BrowserController

@synthesize myWebView, address, backButton, forwardButton, progress;

- (void) resetButtons {
    [self.backButton setEnabled:[self.myWebView canGoBack]];
    [self.forwardButton setEnabled:[self.myWebView canGoForward]];
}

- (void)awakeFromNib {
    [self.address setStringValue:@"http://pragprog.com"];
    [self.myWebView takeStringURLFrom:self.address];
}

- (void)webView:(WebView *)sender
didReceiveTitle:(NSString *)title
    forFrame:(WebFrame *)frame {
    [[sender window] setTitle:title];
}
```

```
- (void)webView:(WebView *)sender
        didFinishLoadForFrame:(WebFrame *)frame {
    [self.address setStringValue:[sender mainFrameURL]];
        [self resetButtons];
►   [self.progress stopAnimation:self];
}

- (void)webView:(WebView *)sender
        didStartProvisionalLoadForFrame:(WebFrame *)frame {
►   [self.progress startAnimation:self];
}
@end
```

Das ist nicht viel Code. Im nächsten Kapitel sehen wir uns an, welche Änderungen notwendig sind, um unseren SimpleBrowser für das iPhone zu implementieren.

Kapitel 10

Unseren Browser für das iPhone anpassen

Obwohl die in diesem Buch vorgestellten Cocoa-Konzepte sowohl für Mac OS X als auch für das iPhone gelten, werden Ihnen einige Unterschiede in den APIs und der Anwendung auffallen. Sie werden das in diesem Kapitel sehen, wenn wir unseren Browser für das iPhone und den iPod Touch anpassen.[1] Ich verwende in diesem Buch das iPhone 3.*x*-SDK. Sie können sich das SDK kostenlos herunterladen, aber zuerst müssen Sie den Nutzungsbedingungen von Apple unter *http://developer.apple.com/iphone* zustimmen.

Es gibt drei Gründe, unseren Browser auf das iPhone zu portieren. Erstens greifen die iPhone-APIs die neuen Features von Objective-C 2.0 auf (z. B. Eigenschaften). Zweitens sind Delegates ein wichtiger Grund dafür, dass die iPhone-APIs sauberer sind. Und drittens ermöglicht es Ihnen die Portierung des Browsers, die einzelnen Schritte noch einmal durchzugehen. Diesmal haben Sie aber eine bessere Vorstellung davon, wo die Reise eigentlich hingeht. Dieser zweite Durchlauf der Entwicklung eines Cocoa-Projekts soll Ihnen dabei helfen, alle Aspekte zusammenzuführen.

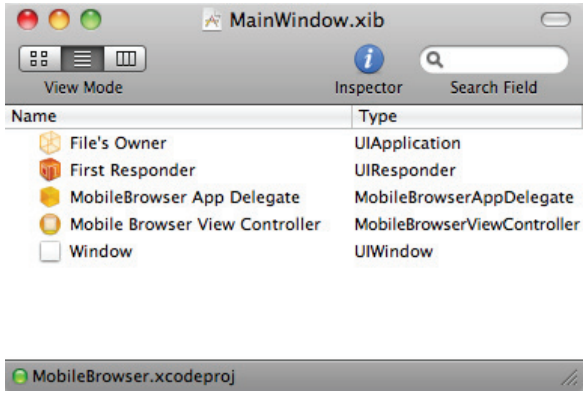
10.1 Das iPhone-Projekt anlegen

Legen wir ein neues Projekt in Xcode an. Diesmal wählen wir `iPhone OS > Application > View-based Application` und nennen es *MobileBrowser*. Führen Sie den MobileBrowser gleich aus, indem Sie *Build & Run*

1 Da die beiden das Gleiche sind, spreche ich kollektiv vom „iPhone“.

in Xcode anklicken. Der Code sollte kompiliert werden, und der iPhone-Simulator sollte starten. Das Hauptfenster sollte einen einzelnen View mit grauem Hintergrund zeigen.

Etwas formaler ausgedrückt, wird beim Start der Browseranwendung eine Instanz der Klasse `UIApplication` erzeugt und das `MainWindow`-Nib geladen.[2]



Das Nib enthält ein die Anwendung repräsentierendes Objekt, dessen Delegate-Outlet mit `MobileBrowserAppDelegate` verknüpft ist. Die *.nib*-Datei enthält außerdem den `MobileBrowserViewController`. In der `applicationDidFinishLaunching:`-Methode des App-Delegate wird dem Fenster der View des View-Controllers hinzugefügt.

iPhoneBrowser/MobileBrowser1/Classes/MobileBrowserAppDelegate.m

```
-(void)applicationDidFinishLaunching:(UIApplication *)application {
    // Override point for customization after app launch
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}
```

So funktionieren die meisten iPhone-Apps. Sie besitzen ein einzelnes Fenster für die gesamte Anwendung. Was Sie als Bildschirm wahrnehmen, ist der Inhalt eines View. Und wenn Sie mit der Reiterleiste, der Navigationsleiste oder irgendeiner anderen Möglichkeit von Bildschirm zu Bildschirm wechseln, dann ersetzen Sie den alten View des Fensters durch einen neuen View, indem Sie sich durch die entsprechenden View-Controller hangeln.

2 iPhone-spezifische Klassen beginnen häufig mit `UI` und nicht wie bei Desktopanwendungen mit `NS`.

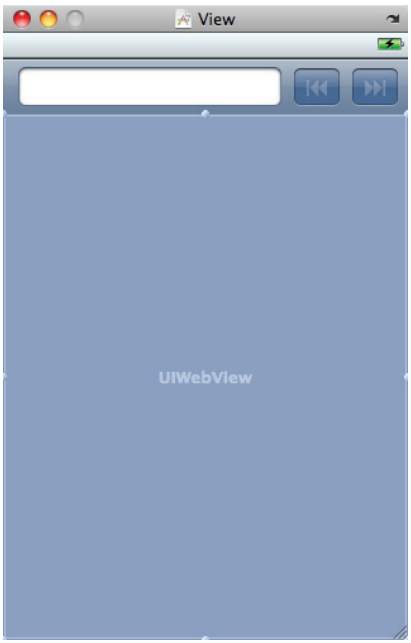
Im Allgemeinen steht jeder dieser Top-Level-Views in einer eigenen *.nib*-Datei, die dem View-Controller gehört. Über den Besitzer einer Datei wollen wir jetzt nicht weiter reden. Wir sagen einfach, dass ein Großteil der Arbeit für die Anwendung in der zweiten *.nib*-Datei und im View-Controller erledigt wird.

Die Aufgabe eines View-Controller besteht darin (Sie ahnen es vielleicht), den View zu kontrollieren. Während es im View um das Aussehen geht, definiert der Controller das Verhalten. Wenn Sie (wie in der `applicationDidFinishLaunching:`-Methode gesehen) einen View in einem Fenster als Subview einfügen möchten, fragen Sie den View-Controller zuerst, welchen View er denn kontrolliert. Sobald alles angerichtet ist, können Sie das Fenster und seinen Inhalt für den Benutzer freigeben und Klicks und andere Eingaben akzeptieren.

Wir werden das Verhalten der Anwendung nicht während des Starts verändern, also richten wir unsere Aufmerksamkeit bei der Entwicklung unseres Webbrowsers auf die *.nib*-Datei, die den View und seinen Controller enthält.

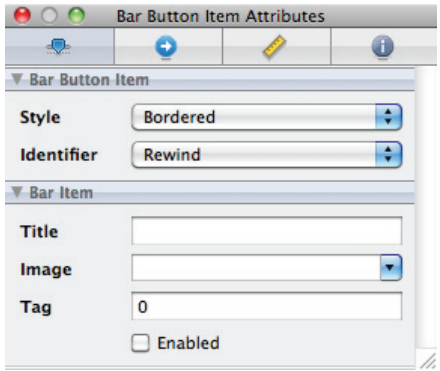
10.2 Das Aussehen unseres Browsers entwickeln

Klicken Sie `MobileBrowserViewController.xib` doppelt an, um es im IB zu öffnen. Wir wollen unseren leeren View in Folgendes verwandeln:



Wie bisher können Sie Komponenten aus der Library ziehen und im View positionieren. Platzieren Sie einen `UIToolbar` oben im View. Ziehen Sie ein `UITextField` links neben das `UIBarButtonItem`, das links neben dem `UIToolbar` liegt. Ziehen Sie ein weiteres `UIBarButtonItem` rechts neben das Textfeld. In unserer Desktopversion hatten wir Platz für die Wörter *Forward* und *Back*. Hier borgen wir uns Images aus, die für Audioschnittstellen gedacht sind. Öffnen Sie den Inspector.

Er unterscheidet sich ein wenig von dem, was Sie aus der Entwicklung für Mac OS X-Anwendungen kennen. Es gibt vier Reiter, die Sie benutzen, um die Attribute, Verbindungen, Größen und Identitäten zu untersuchen und zu ändern. Wählen Sie den Attributes Inspector für den *Zurück*-Button.



Benutzen Sie das Pull-down-Menü, um *Identifier* auf *Rewind* zu setzen. Und wo Sie gerade dabei sind, deaktivieren Sie die *Enabled*-Checkbox, damit der Benutzer den *Zurück*-Button am Anfang nicht anklicken kann. Im gleicher Weise setzen Sie den Identifier des *Vor*-Buttons auf *Fast Forward* und deaktivieren die *Enabled*-Checkbox.

Abschließend wählen Sie einen `UIWebView` und positionieren ihn so, dass er den gesamten View unter dem Toolbar einnimmt. Öffnen Sie den Attributes Inspector für den Web-View und aktivieren Sie die Checkbox bei `Web View > Scales Page to Fit`. Speichern Sie ab. Das ist jetzt erst mal unser Interface.

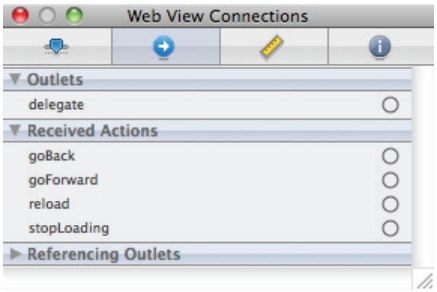
Klicken wir auf *Build & Run* und sehen uns an, was passiert. Unsere Anwendung sollte normal starten. Die *Vor*- und *Zurück*-Buttons sind deaktiviert. Versuchen Sie, eine URL einzugeben. Klicken Sie das Textfeld an. Das sieht vielversprechend aus. Das Textfeld wird gelöscht und die Tastatur erscheint. Sie können eine neue URL eingeben.

Was nun?

Sie drücken ↵, und nichts passiert. Wenn Sie einen anderen Bereich des Bildschirms anklicken, verschwindet die Tastatur. Aber wenn Sie das Textfeld erneut auswählen, wird die URL gelöscht und Sie müssen sie neu eingeben. Wie können Sie eine URL eingeben und sich zur entsprechenden Site bewegen? Es stellt sich heraus, das wir zwei weitere Delegates benötigen.

10.3 Einschränkungen des WebView

Ein `UIWebView` kann nicht so viel wie der `WebView`, mit dem wir es bisher zu tun hatten. Sie erkennen das im Connections Inspector für unseren `UIWebView`.



Vieles fehlt, doch was uns unmittelbar betrifft, ist die Tatsache, dass ein `WebView` eine `takeStringURLFrom:`-Nachricht empfangen kann, ein `UIWebView` hingegen nicht. Das stellt ein Problem dar. In der Desktopversion konnten wir den Web-View mit dem Textfeld verbinden und die URL ohne irgendwelchen Code übergeben. Nun müssen wir einen anderen Weg finden.

10.4 Eine Webseite beim Start laden

Wir können das Textfeld und den Web-View nicht direkt miteinander verbinden, weshalb beide mit unserem Controller verbunden werden müssen. Fügen Sie zwei Outlets in die Header-Datei von `MobileBrowserViewController` ein.

iPhoneBrowser/MobileBrowser1/Classes/MobileBrowserViewController.h

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

@interface MobileBrowserViewController : UIViewController {
    UIWebView *webView;
    UITextField *address;
}
```

```
@property(nonatomic,retain) IBOutlet UIWebView *webView;
@property(nonatomic, retain) IBOutlet UITextField *address;

@end
```

Im Gegensatz zur Desktopversion ist `UIWebView` Teil des iPhone-Frameworks, das wir zu Beginn dieser Datei importieren. Wir müssen daher keinen speziellen Import und für unser Ziel auch kein Framework einbinden.

Auch die Eigenschaften sind andere. Weil wir für das iPhone entwickeln, fügen wir unseren Eigenschaften das `nonatomic`-Attribut hinzu, da wir für `UIKits` Single-Thread-/Single-Core-Modell entwickeln. Uns steht keine Garbage Collection zur Verfügung, weshalb wir `retain` statt `assign` als Speicherattribut verwenden. Denken Sie daran, diese Eigenschaften in der Implementierungsdatei zu synthetisieren.

Öffnen Sie die *.nib*-Datei des `MobileBrowserViewController` und verwenden Sie den Connections Inspector, um diese Outlets vom Dateieigentümer (*File's Owner*) mit den entsprechenden Komponenten im View zu verknüpfen.[3] Speichern Sie und beenden Sie den Interface Builder.

In Xcode wollen wir eine Methode entwickeln, die eine als String übergebene URL laden kann. Gehen wir die Dokumentation für `UIWebView` durch uns schauen wir mal, ob wir nicht eine Methode finden, die das macht, was wir wollen.

Hmmm. Wenn ich unter `Tasks > Loading Content` nachsehe, dann passt Folgendes noch am ehesten:

```
- (void)loadRequest:(NSURLRequest *)request
```

Das ist recht nah dran, verlangt aber einen `NSURLRequest` als Parameter, und keinen String. Wir folgen dem Link zur `NSURLRequest`-Dokumentation und schauen, ob wir dort etwas finden. Diesmal sehe ich mir `Tasks > Creating Requests` an und finde Folgendes:

```
+ (id)requestWithURL:(NSURL *)theURL
```

Es sieht so aus, als wären wir unserem Ziel näher gekommen, doch nun müssen wir herausfinden, wie man einen `NSURL` aus einem String erzeugt. Folgen Sie dem Link zur `NSURL`-Dokumentation und sehen Sie unter `Tasks > Creating an NSURL` nach. Da ist es!

```
+ (id)URLWithString:(NSString *)URLString
```

3 Wir wissen aufgrund des Typs, dass wir File's Owner verwenden müssen.

Diese Methoden benutzen wir nun für unsere loadURLFromTextField-Methode.[4] Fügen Sie Folgendes gleich hinter der @synthesize-Zeile in MobileBrowserViewController.m ein:

iPhoneBrowser/MobileBrowser1/Classes/MobileBrowserViewController.m

```
-(void) loadURLFromTextField {
    NSURL *url = [NSURL URLWithString:self.address.text];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    [self.webView loadRequest:request];
}
```

Wir erzeugen die URL mithilfe der text-Eigenschaft unseres Textfeldes. Wir erzeugen den URL-Request aus der URL und senden dann die Nachricht an das webView-Outlet.

Wir sind fast da – wir brauchen nur noch jemanden, der loadURLFromTextField aufruft, nachdem die Anwendung gestartet wurde. In der Desktopversion haben wir dazu die Methode awakeFromNib verwendet. Bei der iPhone-Entwicklung geschieht es oft als Teil der viewDidLoad-Methode.[5] Wenn Sie sich in Ihrer Implementierungsdatei umsehen, werden Sie entdecken, dass die meisten Methoden auskommentiert sind. Entfernen Sie die Kommentare und fügen Sie die folgende Zeile ein, um die URL zu laden:

iPhoneBrowser/MobileBrowser1/Classes/MobileBrowserViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.address.text = @"http://pragprog.com";
    [self loadURLFromTextField];
}
```

Klicken Sie auf *Build & Run*. Die Anwendung sollte starten und die Pragmatic-Homepage sollte (passend skaliert) in den Browser geladen werden. Sie können sich durch Seiten bewegen, wenn Sie Links anklicken, doch Sie können keine URLs eingeben. Darum wollen wir uns als Nächstes kümmern.

10.5 Das Textfeld im IB einstellen

So leistungsfähig und cool das iPhone auch ist – es ist doch vielen Einschränkungen unterworfen. Für die meisten von uns ist es einfacher,

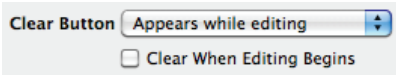
4 Beachten Sie, dass wir Klassenmethoden verwenden, um Autorelease-Instanzen des Requests und der URL zu erzeugen. Wir müssen sie nicht selbst freigeben.

5 Andere Methoden, die wir für diese Arbeiten häufig benutzen, sind viewWillAppear und viewDidAppear.

Text an unseren Laptops einzugeben als im iPhone. Zwar ist das Display des iPhone sehr schön und die Auflösung gut, auf unseren Desktopmonitoren und auf Laptops haben wir aber doch deutlich mehr Platz. Wenn wir für das iPhone entwickeln, müssen wir die Beschränkungen des Geräts berücksichtigen und es unseren Benutzern so leicht wie möglich machen, mit der Anwendung zu interagieren.

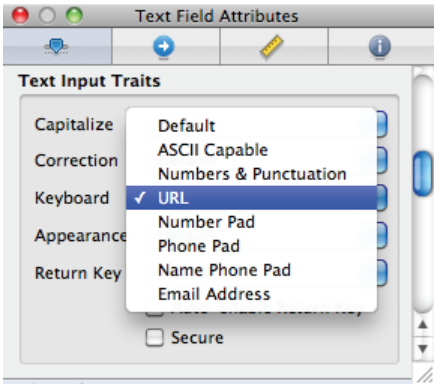
Lassen Sie uns das Textfeld und die Tastatur so konfigurieren, das es für die Benutzer einfacher wird, URLs einzugeben. Öffnen Sie das `MobileBrowserViewController`-Nib und sehen Sie sich das Textfeld mit dem Attributes Inspector an.

Das Erste, was mich bei der Arbeit mit diesem Textfeld stört, ist die Tatsache, dass sein Inhalt immer gelöscht wird, wenn ich hineinklicke. Ich würde mich gerne entscheiden können, ob ich das Feld leeren oder es editieren möchte, wenn ich mich bei ein oder zwei Buchstaben vertippt habe. Deaktivieren Sie die Box *Clear When Editing Begins* und nutzen Sie das Pull-down-Menü, um den *Clear Button* auf *Appears while editing* zu setzen.



Klicken Sie *Build & Run* an. Sie erkennen nun beim Editieren des Feldes ein kleines `x` auf der rechten Seite. Das Textfeld wird nicht mehr gelöscht, wenn Sie es auswählen, und Sie können den Text doppelklicken und die Ausschneiden- und Einfügen-Funktionen des iPhone verwenden.

Sehen Sie sich die Texteingabe-Möglichkeiten (Text Input Traits) im Attributes Inspector an. Wir wollen eine Tastatur wählen, die für die Webnavigation besser geeignet ist. Hier die verfügbaren Optionen:



Wählen Sie *URL*. Das ist eine kleine, aber sehr wichtige Änderung. Sie stellt die Tasten zur Verfügung, die die URL-Eingabe vereinfachen und so die Nutzbarkeit für den Anwender stark verbessern. Als letzte Korrektur ändern wir den *Return Key* von *Default* zu *Go*. Speichern Sie ab und führen Sie die Anwendung erneut aus. Nun sieht die Tastatur so aus:

10.6 Den Textfeld-Delegate nutzen

Sie haben beim Textfeld bisher nur das Aussehen und die mit ihm verknüpfte Tastatur geändert. Aber die virtuelle Tastatur weiß nicht, wie sie reagieren soll, wenn der Benutzer Daten eingegeben und den *Go*-Button angeklickt hat. Dafür benötigen wir einen Delegate.[6]

Im Interface Builder verwenden Sie den Connections Inspector, um das `delegate`-Outlet des Textfeldes mit dem *File's Owner* zu verknüpfen. Speichern Sie ab. In Xcode fügen Sie die Deklaration des `UITextFieldDelegate` in spitzen Klammern in die `MobileBrowserViewController`-Header-Datei ein:

6 Bei `NSTextField` arbeiten Sie mit einer physischen Tastatur und benötigen keinen Delegate.

iPhoneBrowser/MobileBrowser3/Classes/MobileBrowserViewController.h

```
@interface MobileBrowserViewController : UIViewController
                                          <UITextFieldDelegate> {
```

Damit haben wir der Welt mitgeteilt, dass unser `MobileBrowserViewController` weiß, wie er auf jede der sieben optionalen Nachrichten zu Beginn und am Ende des Editierens reagieren muss, und was bei Return, beim Löschen des Feldes und bei der Änderung einiger Zeichen zu tun ist.

Lassen Sie uns beispielsweise den Web-View deaktivieren, wenn der Benutzer damit beginnt, die URL zu editieren:

iPhoneBrowser/MobileBrowser3/Classes/MobileBrowserViewController.m

```
-(void)textFieldDidBeginEditing:(UITextField *)textField {
    [self disableWebView];
}
```

Um den Web-View zu deaktivieren, hindern wir den Benutzer daran, mit ihm zu arbeiten, und dimmen ihn ein wenig, indem wir seinen Alphawert verändern. Damit machen wir visuell deutlich, dass er momentan nicht aktiv ist.[7] Wir aktivieren ihn wieder, indem wir die obigen Schritte rückgängig machen.

iPhoneBrowser/MobileBrowser3/Classes/MobileBrowserViewController.m

```
-(void) disableWebView {
    self.webView.userInteractionEnabled = NO;
    self.webView.alpha = 0.25;
}
-(void) enableWebView {
    self.webView.userInteractionEnabled = YES;
    self.webView.alpha = 1.0;
}
```

Klickt der Benutzer auf der Tastatur *Go* an, wird die URL geladen und wir weisen das Textfeld über `resignFirstResponder` an, die Tastatur verschwinden zu lassen. All das packen wir in die Delegate-Methode `textField-ShouldReturn:`.

iPhoneBrowser/MobileBrowser3/Classes/MobileBrowserViewController.m

```
-(BOOL) textFieldShouldReturn:(UITextField *)textField {
    [self loadURLFromTextField];
    [textField resignFirstResponder];
    [self enableWebView];
    return YES;
}
```

7 Beachten Sie, dass `alpha` eine Eigenschaft von `UIView` ist; Sie können diesen Wert also für jede Subklasse von `UIView` nutzen, nicht nur für `UIWebView`.

Klicken Sie auf *Build & Run*. Sie sollten URLs eingeben und über *Go* die entsprechenden Seiten laden können. Unser Browser funktioniert nun größtenteils. Als Nächstes wollen wir noch die Buttons ans Laufen bringen.

10.7 Ein dritter Delegate zur Implementierung der Buttons

Wenn wir uns unseren `BrowserController` noch einmal anschauen, merken wir, dass wir wissen müssen, wann das Laden einer Webseite abgeschlossen ist, um die Buttons zurücksetzen zu können. Die einzige Möglichkeit herauszufinden, wann die Seite geladen wurde, ist die Verwendung von `UIWebView-Delegate`.

Wir verwenden unseren `MobileBrowserViewController` als Delegate für den Web-View. Wir fügen außerdem Outlets für die *Forward*- und *Back*-Buttons ein. Unsere Header-Datei sieht nun so aus:

iPhoneBrowser/MobileBrowser4/Classes/MobileBrowserViewController.h

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

@interface MobileBrowserViewController : UIViewController
                                            <UITextFieldDelegate> {
    UIWebView *webView;
    UITextField *address;
    UIBarButtonItem *backButton;
    UIBarButtonItem *forwardButton;
}
@property(nonatomic,retain) IBOutlet UIWebView *webView;
@property(nonatomic, retain) IBOutlet UITextField *address;
@property(nonatomic, retain) IBOutlet UIBarButtonItem *backButton;
@property(nonatomic, retain) IBOutlet UIBarButtonItem *forwardButton;

@end
```

Öffnen Sie den `MobileBrowserViewController`-Nib im Interface Builder. Wir müssen fünf Verbindungen herstellen.

1. Wählen Sie den *File's Owner* und verknüpfen Sie das `backButton`-Outlet mit dem *Back*-Button und das `forwardButton`-Outlet mit dem *Forward*-Button.

2. Wählen Sie den Web-View und verbinden Sie seinen Delegate mit dem *File's Owner*.

3. Immer noch im Web-View verbinden Sie die Aktion `goBack` mit dem *Back*- und `goForward` mit dem *Forward*-Button.

Speichern Sie Ihre Arbeit. Nun wollen wir den Button-Reset genau so implementieren, wie wir es für den Desktopbrowser getan haben. Vergessen Sie nicht, die Eigenschaften zu synthetisieren. Als Nächstes wollen wir eine von uns entwickelte Methode namens `resetButtons:` aus der Delegate-Methode aufrufen, die uns darüber informiert, dass die Webseite geladen wurde. Beim iPhone heißt diese Methode `webViewDidFinishLoad:`.

iPhoneBrowser/MobileBrowser4/Classes/MobileBrowserViewController.m

```
-(void) resetButtons:(UIWebView *) theWebView {
    [self.backButton setEnabled:[theWebView canGoBack]];
    [self.forwardButton setEnabled:[theWebView canGoForward]];
}
-(void)webViewDidFinishLoad:(UIWebView *)theWebView {
    [self resetButtons:theWebView];
}
```

Starten Sie den Browser und bewegen Sie sich zu einer zweiten Webseite. Sobald die Seite geladen ist, sollte der *Back*-Button aktiviert sein. Klicken Sie den ihn an. Sie kehren zur vorigen Seite zurück, und nun sollte auch der *Forward*-Button aktiv sein. Wir besitzen nun einen funktionierenden einfachen Browser für das iPhone.

10.8 Übung: Eine Aktivitätsanzeige einfügen

Fügen Sie einen `UIActivityIndicatorView` in Ihre Anwendung ein. Im Interface Builder setzen Sie den Stil (Style) auf *Large White* und aktivieren die Checkbox *Hide When Stopped*.

Die Aktivitätsanzeige soll rotieren, solange die Webseite geladen wird. Der Web-View soll deaktiviert bleiben, bis die Seite vollständig geladen ist. Schließlich soll das Textfeld noch die URL der dargestellten Seite enthalten, und nicht die vom Benutzer zuletzt eingegebene Adresse.

10.9 Lösung: Eine Aktivitätsanzeige einfügen

Zuerst fügen wir folgendes Outlet in die `MobileBrowserViewController`-Header-Datei ein:[8]

8 Denken Sie daran, dass Sie Deklarationen der Instanzvariablen löschen können, wenn der Simulator repariert ist.

iPhoneBrowser/MobileBrowser5/Classes/MobileBrowserViewController.h

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

@interface MobileBrowserViewController : UIViewController
                                                <UITextFieldDelegate>
{
    UIWebView *webView;
    UITextField *address;
    UIBarButtonItem *backButton;
    UIBarButtonItem *forwardButton;
►   UIActivityIndicatorView *activityView;
}
@property(nonatomic,retain) IBOutlet UIWebView *webView;
@property(nonatomic, retain) IBOutlet UITextField *address;
@property(nonatomic, retain) IBOutlet UIBarButtonItem *backButton;
@property(nonatomic, retain) IBOutlet UIBarButtonItem *forwardButton;
@property(nonatomic, retain) IBOutlet UIActivityIndicatorView *activityView;
@end
```

Nun öffnen wir die *.nib*-Datei und ziehen den Aktivitätsanzeiger-View über den Web-View. Verbinden Sie das Outlet mithilfe des Connections Inspector und nutzen Sie dann den Attributes Inspector, um den Stil als *Large White* festzulegen und die Checkbox *Hide When Stopped* zu aktivieren. Speichern Sie ab und kehren Sie zu Xcode zurück.

Wir können alle notwendigen Änderungen in den Methoden `webViewDidStartLoad:` und `webViewDidFinishLoad:` vornehmen. Zusätzlich können wir den Aufruf von `enableWebView:` in `textFieldShouldReturn:` entfernen. Wir wollen kurz innehalten und uns den Code ansehen, den wir in diesem Kapitel entwickelt haben. Ich finde es beeindruckend, dass wir einen Web-View, ein Textfeld mit passender Tastatur, Vor- und Zurück-Buttons und eine Aktivitätsanzeige mit so wenig Code implementiert haben.

iPhoneBrowser/MobileBrowser5/Classes/MobileBrowserViewController.m

```
#import "MobileBrowserViewController.h"

@implementation MobileBrowserViewController

@synthesize address, webView, backButton, forwardButton, activityView;

//View-Controller Utility-Methoden
-(void) disableWebView {
    self.webView.userInteractionEnabled = NO;
    self.webView.alpha = 0.25;
}

-(void) enableWebView {
    self.webView.userInteractionEnabled = YES;
    self.webView.alpha = 1.0;
}
```

```
-(void) loadURLFromTextField {
    NSURL *url = [NSURL URLWithString:self.address.text];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    [self.webView loadRequest:request];
}

-(void) resetButtons:(UIWebView *) theWebView {
    [self.backButton setEnabled:[theWebView canGoBack]];
    [self.forwardButton setEnabled:[theWebView canGoForward]];
}

//Initialisierung
-(void)viewDidLoad {
    [super viewDidLoad];
    self.address.text = @"http://pragprog.com";
    [self loadURLFromTextField];
}

//Web-View Delegate-Methoden
-(void)webViewDidStartLoad:(UIWebView *) theWebView {
    [self disableWebView];
    [self.activityView startAnimating];
}

-(void)webViewDidFinishLoad:(UIWebView *)theWebView {
    [self enableWebView];
    [self.activityView stopAnimating];
    [self resetButtons:theWebView];
    self.address.text = [[self.webView.request URL] absoluteString];
}

//Textfeld Delegate-Methoden
-(void)textFieldDidBeginEditing:(UITextField *)textField {
    [self disableWebView];
}
-(BOOL) textFieldShouldReturn:(UITextField *)textField {
    [self loadURLFromTextField];
    [textField resignFirstResponder];
    return YES;
}

//Speicherverwaltung
-(void)didReceiveMemoryWarning {
       // View freigeben, wenn es keinen Superview gibt.
    [super didReceiveMemoryWarning];
}
-(void)dealloc {
    self.address = nil;
    self.webView = nil;
    self.backButton = nil;
    self.forwardButton = nil;
    self.activityView = nil;
    [super dealloc];
}
@end
```

Wir haben drei Delegates und einige Eigenschaften verwendet, und die längste Methode umfasst gerade einmal vier Zeilen. Ich hoffe, dass dieser zweite Blick auf die Entwicklung eines Webbrowsers (diesmal als iPhone-App) Ihnen dabei hilft, einen eigenen Rhythmus für die Entwicklung einer Cocoa-Anwendung zu finden.

10.10 Organisation mit Pragma Marks

Ich habe Kommentare in das letzte Codelisting eingefügt, um die Methoden in Kategorien zu unterteilen. Das hilft mir dabei, die Methode zu finden, wenn ich den Quellcode durchgehe – aber es gibt noch eine bessere Möglichkeit.

Wenn Sie sich das Editorfenster oben genau ansehen, erkennen Sie mehrere Drop-down-Menüs. Eines enthält alle anderen geöffneten Dateien und erlaubt den schnellen Wechsel zwischen ihnen. Ein anderes zeigt Ihnen alle Methodennamen. Wählen Sie eine aus, und Sie gelangen zur entsprechenden Stelle im Quellcode. Momentan sieht das so aus:

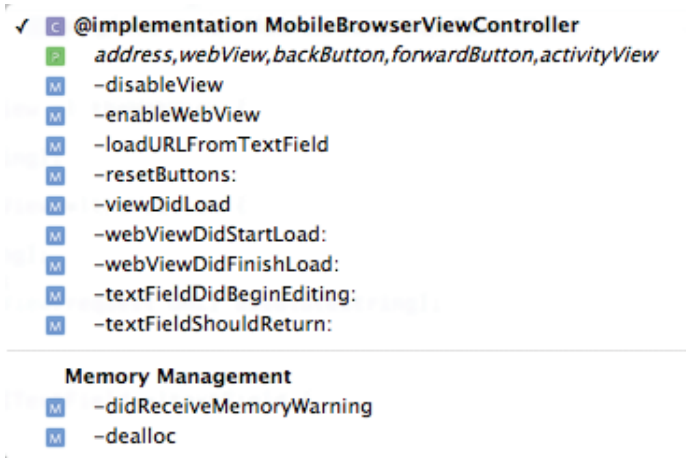
Alles ist bunt zusammengewürfelt. Das ist immer noch besser, als sich durch einen langen Quellcode kämpfen zu müssen, doch mithilfe von `#pragma mark` lässt es sich noch verbessern. Ersetzen Sie den Kommentar

```
//Speicherverwaltung
```

durch

```
#pragma mark -
#pragma mark Speicherverwaltung
```

Jetzt sieht die Drop-down-Liste der Methoden wie folgt aus:

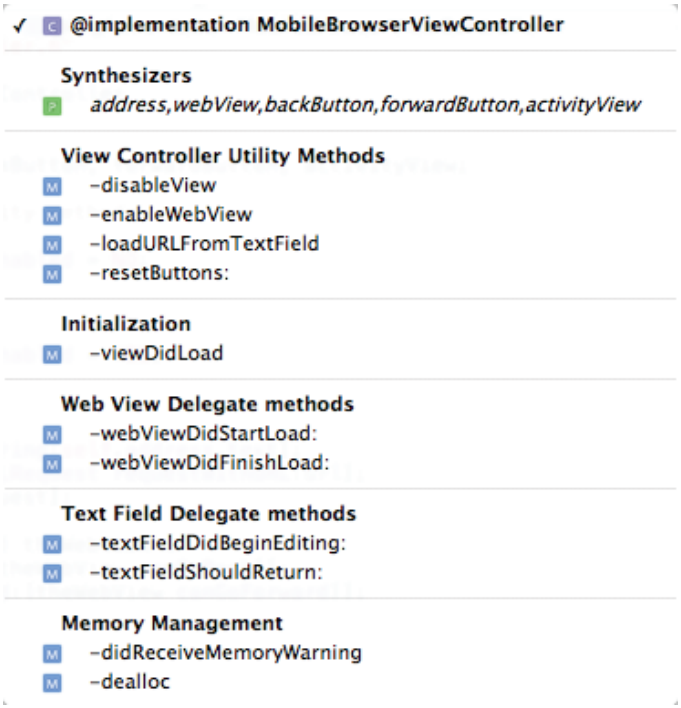


Das `#pragma mark` zeichnet die horizontale Trennlinie, und `#pragma mark Speicherverwaltung` die fett hervorgehobene Überschrift.[9] Führen Sie diese Überschriften für alle Abschnitte ein, und Ihr Code wird sehr viel übersichtlicher, wie die nächste Seite zeigt.[10]

Unsere Webbrowser für den Desktop und das iPhone haben uns gute Dienste geleistet. Ich habe sie benutzt, um Ihnen die Entwicklung von Klassen in Programmcode zu demonstrieren, sowie die Instanziierung von Objekten im Code oder im Nib. Sie haben Methoden mit und ohne Parameter entwickelt und aufgerufen.

9 Hängen Sie keine Leerzeichen hinter dem - an, oder `#pragma mark -` erzeugt anstelle der horizontalen Linie nur einen Strich in der Drop-down-Liste.

10 Wenn Sie jede Eigenschaft in einer eigenen Zeile synthetisiert haben, erscheinen sie auch alle in einer eigenen Zeile, und nicht wie hier alle in einer einzigen Zeile.



Sie haben mit Eigenschaften gearbeitet und gelernt, wie man den Speicher verwaltet. Sie haben Ihre Anwendung mit Aktionen und Outlets verknüpft und die Verwendung von Delegates kennengelernt. Das sind fundamentale Techniken, die Sie als Cocoa-Entwickler ständig nutzen werden.

Doch es gibt noch mehr ...

Kapitel 11

Notifikationen absetzen und abfangen

Es gibt so Vieles, worum man sich im Leben kümmern muss. Es wäre für Sie nicht besonders praktisch, wenn Sie alle paar Minuten (oder auch nur einmal täglich) innehalten und den Verlauf von allem überprüfen müssten. In den meisten Fällen wollen Sie einfach nur wissen, wenn sich etwas geändert hat.

Vielleicht wollen Sie informiert werden, wenn eine Aktie einen bestimmten Preis erreicht hat. Vielleicht soll diese Information aber auch direkt an einen Broker gehen, zusammen mit den passenden Anweisungen. Wie könnte man so etwas hinbekommen?

Sie müssten sich mit der Stelle in Verbindung setzen, die etwas über ein bestimmtes Ereignis weiß. In unserem Fall geht es um so etwas wie „sag mir Bescheid, wenn XXX auf yyy fällt“. Sie müssen dieser Stelle mitteilen, wer die Benachrichtigung erhalten soll. Soll die Notifikation direkt an Sie gehen oder an jemanden, der in Ihrem Auftrag handelt?

Egal ob diese Nachricht nun bei uns oder bei jemand anderem landet, gilt es schnell herauszufinden, was mit ihr geschehen soll. Wir erhalten jeden Tag viele Notifikationen. Es ist für uns wesentlich einfacher, sie zu verarbeiten und den nächsten Schritt anzustoßen, wenn uns der Kontext klar ist. Bei Programmcode geschieht das, indem wir der Notifikationsstelle mitteilen, welche Nachrichten Sie uns senden soll. Die Nachricht könnte `kaufen:` oder `verkaufen:` lauten. Mit anderen Worten teilen wir der Notifikationsstelle die Aktion mit, die sie an das von uns bereits festgelegte Ziel sendet.

Die Notifikationsstelle wird unserem Ziel dann als Teil der Nachricht eine Notifikation senden. Für Programmcode bedeutet das, dass ein Notifikationsobjekt der Parameter für die Nachricht ist. Die Notifikation wird Informationen enthalten, die der Empfänger benötigen könnte. Sie finden immer den Namen der Notifikation und Informationen darüber, wer die Notifikation „gepostet" (also abgesetzt) hat. Sie erhalten möglicherweise auch weitere Informationen, die Sie benötigen, um die Notifikation verarbeiten zu können.[1]

11.1 Übung: Ein Modell aufbauen

Unser Beispielprojekt in den nächsten Kapiteln wird die Namen und Icons der Anwendungen darstellen, die gestartet und beendet werden. Wie bei unserem Browser geht es nicht darum, eine vollständige Anwendung zu entwickeln. Vielmehr geht es darum, mehr über bestimmte Aspekte der Cocoa-Programmierung zu lernen. In diesem Kapitel beginnen wir mit Notifikationen.

Legen Sie eine neue Cocoa-Anwendung an und nennen Sie sie `HelloApplication`. Vergegenwärtigen Sie sich, dass wir uns wieder in der Welt der Mac OS X-Apps bewegen, uns also nicht um die Speicherverwaltung kümmern müssen. Ändern Sie die Projekteinstellungen so, dass die Garbage Collection genutzt wird. Das machen wir von nun an für jedes neue Projekt.

Fügen Sie zwei Klassen hinzu, die beide `NSObject` erweitern. Die Klasse `CurrentApp` wird das Modell dieser Anwendung repräsentieren. Passen Sie sie im Moment noch nicht an. Die Klasse `ActivityController` dient uns als Controller. Instanziieren Sie sie in `MainMenu.xib`. Erzeugen Sie eine Instanz von `CurrentApp` im `awakeFromNib` des `ActivityController` und speichern Sie sie in einer Eigenschaft namens `currentApp`.

11.2 Lösung: Ein Modell aufbauen

Im Moment trainieren Sie Ihr Gehirn nur im Bezug auf die Erzeugung neuer Projekte und Dateien. Sie haben `HelloApplication` erzeugt und dabei alle Checkboxen deaktiviert gelassen. Sie haben eine neue Datei namens `CurrentApp.m` erzeugt, also eine `NSObject` erweiternde Objective-C-Klasse. Sie haben diesen Vorgang für `ActivityController.m` wiederholt.

1 In diesem Kapitel sehen wir uns das Notifikationsobjekt nicht näher an. Unser Beispiel konzentriert sich auf die Registrierung zum Zweck des Empfangens von und Reagierens auf Notifikationen. Wir widmen uns diesem Thema in Kapitel 13, *Mit Dictionaries arbeiten*, und zeigen Ihnen dort, wie man mit dem Notifikationsobjekt arbeitet.

Nun instanziieren Sie Ihre Klassen. Im Interface Builder ziehen Sie eine Instanz von `ActivityController` in das Dokumentenfenster von `MainMenu.xib`.

In Xcode deklarieren Sie eine Instanzvariable und eine Eigenschaft namens `currentApp` vom Typ `CurrentApp` in `ActivityController.h`. Denken Sie daran, `@class` zu verwenden, um die `CurrentApp`-Klasse im Vorfeld zu deklarieren.

Warnung

Bald wird ein Problem bei diesem Projekt auftauchen, weil die `ActivityController`- und `CurrentApp`-Objekte direkt freigegeben werden, sobald sie erzeugt wurden. Das ist im Moment nicht weiter schlimm, aber bald werden wir sie behalten wollen, solange die Anwendung läuft. Das ist ein subtiles und ernstes Problem, das man möglicherweise nur schwer erkennt. Wir werden die Objekte festhalten, nachdem Sie das Problem in Aktion gesehen haben.

Notication/HelloApplication1/ActivityController.h

```
#import <Cocoa/Cocoa.h>
@class CurrentApp;

@interface ActivityController : NSObject {
    CurrentApp *currentApp;
}
@property CurrentApp *currentApp;

@end
```

Ich hoffe, Sie haben daran gedacht, die Vorwärtsdeklaration für `CurrentApp` gleich nach der `import`-Anweisung einzufügen. Die Property-Deklaration sieht ein wenig nackt aus. Ohne Gargabe Collection müsste das Speicherattribut `retain` lauten. Mit der Garbage Collection entspricht das `assign`, und `assign` ist der Standardwert, weshalb wir ihn ganz weglassen können.[2]

In `ActivityController.m` müssen Sie `currentApp` synthetisieren und eine entsprechende Instanz in `awakeFromNib` erzeugen. Denken Sie daran, `CurrentApp.h` zu importieren.

2 Wenn Sie für 64-Bit-Maschinen entwickeln, können Sie natürlich auch die Instanzvariable weglassen.

Notication/HelloApplication1/ActivityController.m

```
#import "ActivityController.h"
#import "CurrentApp.h"

@implementation ActivityController

@synthesize currentApp;

-(void)awakeFromNib {
    self.currentApp = [[CurrentApp alloc] init];
}
@end
```

11.3 Für Notifikationen registrieren

Ein Objekt wird zum Empfangen von Notifikationen wie folgt registriert:

```
[[NSNotificationCenter defaultCenter]
               addObserver:observerObject
                  selector:@selector(methodName:)
                      name:NameOfNotification
                    object:notifyingObject];
```

Ein, zwei, viele Objekte können sich bei einem Notification-Server für die gleiche Notifikation registrieren, und ein Objekt kann sich für so viele unterschiedliche Arten von Notifikationen registrieren, wie es will. Sehen wir uns die Argumente der Registrierung genauer an:

- Sie senden die Nachricht `addObserver:selector:name:object:` an das Standard-Notifikationszentrum.
- In unserem Fall geben wir das `observerObject` mit `self` an, aber wir können jedes Objekt angeben, das die Notifikation empfangen will. Dieses Objekt ist das Ziel der Aktion, die Sie durch die Notifikation anstoßen.
- Der `selector` legt die Aktion fest. Das ist der Name der zum Observer gehörenden Methode, die beim Empfang der Notifikation ausgeführt wird.
  - Der Name dieser Methode endet mit einem Doppelpunkt, weil die Methode einen einzelnen Parameter verlangt.
  - Dieser einzelne Parameter ist das Notifikationsobjekt, das beim Senden der Notifikation übergeben wird.
- Der `name` ist der Name der Notifikation, für die man sich registriert hat.

- Das `object` verweist auf das Objekt, an dessen Notifikationen Sie interessiert sind. In unserem *SimpleBrowser*-Beispiel könnte es mehr als einen Web-View geben, der Notifikationen sendet, während Sie nur einen bestimmten davon überwachen wollen. Sie können auch `nil` übergeben, um die Notifikation von allen Objekten zu empfangen, die diese Notifikation in Ihrer Anwendung nutzen.

Bei unserem Börsenbeispiel sind Sie oder Ihr Stellvertreter das `observerObject`. Der `selector` ist die auszuführende Aktion wie `kaufen:` oder `verkaufen:`. Der Name ist die Notifikation, für die Sie registriert sind, etwa „Preis fällt auf yyy“. Das `Objekt` könnte der Name der Aktie sein, die Sie beobachten.

Das Prinzip besteht darin, dass Sie sich für eine Notifikation registrieren, indem Sie den Namen einer Methode übergeben, die für ein bestimmtes Objekt aufgerufen wird. Wenn die Methode aufgerufen wird, empfängt sie ein `NSNotication`-Objekt mit einem einzigen Parameter.

Jedes Notifikationsobjekt enthält zwei oder drei Informationen, einschließlich des Namens der Notifikation und des Objekts, das die Notifikation initiiert hat. Muss die Notifikation mehr Informationen übergeben, speichert sie diese in einem dritten Objekt: einem `NSDictionary` namens `userInfo`.

Die drei `NSNotication`-Instanzmethoden sind im Grunde die Getter für diese Eigenschaften:

```
-(NSString *)name
-(id)object
-(NSDictionary *)userInfo
```

Die `name`-Methode gibt den Namen der Notifikation zurück und die `object`-Methode ein Handle auf das Objekt, das die Notifikation angestoßen hat. Die Methode `userInfo` gibt Schlüssel/Wert-Paare mit notifikationsspezifischen Informationen zurück. Wir sehen uns `userInfo` in Kapitel 13, *Mit Dictionaries arbeiten*, auf Seite 215 an.

Im nächsten Abschnitt konkretisieren wir das Ganze. Wir registrieren uns für Notifikationen, die Änderungen im Status unseres Workspace anzeigen.

11.4 Auf Workspace-Aktivitäten reagieren

Ihre Anwendung wird in einem Arbeitsbereich ausgeführt, dem sogenannten Workspace. Wir können verfolgen, was im Workspace vor sich geht, indem wir entsprechende Notifikationen abfangen. Gegen Ende des Kapi-

tels werden wir uns auf Notifikationen beschränken, die den Start und das Ende von Anwendungen betreffen. Beginnen wollen wir aber damit, *alle* Notifikationen abzufangen, die vom `NSWorkspace` gesendet werden.

Wir verwenden eine Klassenmethode, um den Singleton zu bestimmten, der unseren gemeinsam genutzten Workspace repräsentiert, und fordern von diesem dann eine Referenz auf sein Notification-Center an:

```
NSNotificationCenter *defaultCenter = [[NSWorkspace sharedWorkspace]
                                                    notificationCenter];
```

Als Nächstes registrieren wir uns, um die Notifikationen zu empfangen, die uns interessieren:

```
[defaultCenter addObserver:self
                  selector:@selector(reportActivity:)
                      name:nil
                    object:nil];
```

Wir hätten um die Benachrichtigung bei einem bestimmten Ereignis bitten können, wenn wir es im Parameter `name:` übergeben hätten. Doch wir haben `nil` übergeben, weshalb uns das Notification-Center über alle Ereignisse informiert. Die Kombination aus `self` für den Observer und `reportActivity:` für den Selektor legt das Ziel und die Aktion fest, die aufgerufen wird, wenn eine Notifikation empfangen wird. Die Notifikation wird als Parameter gesendet.

Hier das vollständige Listing für `CurrentApp.m`:

Notication/HelloApplication2/CurrentApp.m

```
#import "CurrentApp.h"

@implementation CurrentApp

-(void) reportActivity: (NSNotification *) notification {
    NSLog(@"%@", notification.name );
}
-(void) registerNotifications {
   NSNotificationCenter *defaultCenter = [[NSWorkspace sharedWorkspace]
                                                    notificationCenter];
        [defaultCenter addObserver:self
                          selector:@selector(reportActivity:)
                              name:nil
                            object:nil];
}
-(id) init {
    if (self =[super init]) {
        [self registerNotifications];
    }
    return self;
}
@end
```

In `reportActivity:` geben wir nur den Namen der Notifikation im Konsolenfenster aus. Klicken Sie auf *Build & Run*. Was passiert?

Ein Fenster wird für die Anwendung geöffnet, und ich sehe auch so etwas wie im Konsolenfenster.

```
NSWorkspaceDidLaunchApplicationNotification
```

Zum Spaß starte ich iCal. Ich sollte eine weitere `NSWorkspaceDidLaunchApplicationNotication` im Konsolenfenster sehen, aber das ist nicht der Fall.

Ich beende iCal. Ich sollte `NSWorkspaceDidTerminateApplicationNotication` in der Konsole sehen, doch auch das ist nicht der Fall. [3]

Grrrrrr! Das ist das vorhin angesprochene Speicherproblem. Lassen Sie uns eine Pause einlegen und den Fehler beheben.

11.5 Am Controller festhalten

Wir haben unsere Instanz von `CurrentApp` als Observer für Notifikationen eingerichtet. Das einzige Objekt, das eine feste Referenz zu diesem Objekt besitzt, ist unsere Instanz des `ActivityController`. Der `ActivityController` wurde in der *.nib*-Datei angelegt, aber bisher interessiert sich niemand für ihn.[4]

Lassen Sie uns im App-Delegate ein Outlet hinzufügen:

Notication/HelloApplication3/HelloApplicationAppDelegate.h

```
#import <Cocoa/Cocoa.h>
► @class ActivityController;

@interface HelloApplicationAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
►   ActivityController *ac;
}
@property IBOutlet NSWindow *window;
► @property IBOutlet ActivityController *ac;
@end
```

3 Wenn Sie weitere Notifikationen sehen, überprüfen Sie die Projekteinstellungen und stellen Sie sicher, dass die Garbage Collection auf *Required* gesetzt ist.

4 Auch wenn es mich meinen Ruf kostet – das erinnert mich an eine Szene aus *Superman*, wo Superman Lois Lane auffängt und sagt: „Keine Sorge, ich habe Sie." Sie sieht verwirrt aus und fragt: „Aber wer hat Sie?"

Wechseln Sie in die Implementierungsdatei und synthetisieren Sie ac:

Notication/HelloApplication3/HelloApplicationAppDelegate.m

```
#import "HelloApplicationAppDelegate.h"

@implementation HelloApplicationAppDelegate

► @synthesize window, ac;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
}
@end
```

Speichern Sie ab und öffnen Sie `MainMenu.xib` im Interface Builder. Verbinden Sie das gerade erzeugte Outlet mit dem `ActivityController`. Speichern Sie Ihre Arbeit und klicken Sie auf *Build & Run*.

Wenn Sie iCal jetzt starten und wieder beenden, sieht das Konsolenfenster wie folgt aus:

```
NSWorkspaceDidLaunchApplicationNotification
NSWorkspaceWillLaunchApplicationNotification
NSWorkspaceDidActivateApplicationNotification
NSWorkspaceDidDeactivateApplicationNotification
NSWorkspaceDidLaunchApplicationNotification
NSWorkspaceDidActivateApplicationNotification
NSWorkspaceDidDeactivateApplicationNotification
NSWorkspaceDidTerminateApplicationNotification
NSWorkspaceDidActivateApplicationNotification
NSWorkspaceDidDeactivateApplicationNotification
```

Es passiert so einiges, und unsere `ActivityController`- und `CurrentApp`-Objekte existieren jetzt lange genug, um uns davon berichten zu können. Zuerst startet unser `HelloApplication`-Projekt und wir erhalten eine entsprechende Notifikation. Wenn ich iCal starte, erhalte ich die Notifikation, dass iCal starten wird und dann dass iCal gestartet wurde. Dazwischen finden sich Notifikationen über aktivierte und deaktivierte Anwendungen. Nun wollen wir unseren Code so anpassen, dass er sich auf die von uns gewünschten Informationen konzentriert: das Starten und Beenden von Anwendungen.

11.6 Übung: Für Notifikationen registrieren

Modifizeren Sie den Code so, dass `NSWorkspaceDidLaunchApplicationNotication` und `NSWorkspaceDidTerminateApplicationNotication` erkannt werden. Geben Sie dann „Launched“ oder „Terminated“ im Konsolenfenster aus.

11.7 Lösung: Für Notifikationen registrieren

Wir wollen mit der Registrierung und der Verarbeitung von Startnotifikationen beginnen:

Notication/HelloApplication4/CurrentApp.m

```
-(void) registerNotifications {
  NSNotificationCenter *defaultCenter = [[NSWorkspace sharedWorkspace]
                                                        notificationCenter];
 [defaultCenter addObserver:self
►                   selector:@selector(applicationDidLaunch:)
►                       name:NSWorkspaceDidLaunchApplicationNotification
                     object:nil];
}
```

Der Hauptunterschied besteht darin, dass wir den Namen der gewünschten Notifikation angeben, statt über `nil` alle Notifikationen abzufangen. Ich habe darüber hinaus den Namen der Callback-Methode in `applicationDidLaunch:` geändert.

Wenn ich dem gleichen Muster folge, um Beendigungsnotifikationen abzufangen und auf diese zu reagieren, erzeuge ich sehr viel doppelten Code.

Ich führe daher eine Hilfsmethode namens `setUpNotication:withSelector:` und refaktoriere das Ganze so:

Notication/HelloApplication5/CurrentApp.m

```
#import "CurrentApp.h"

@implementation CurrentApp

-(void) applicationDidLaunch: (NSNotification *) notification {
    NSLog(@"Launched.");
}

-(void) applicationDidTerminate: (NSNotification *) notification {
    NSLog(@"Terminated.");
}

-(void)setUpNotification:(NSString *)notification withSelector:(SEL)methodName {
    [[[NSWorkspace sharedWorkspace] notificationCenter]
         addObserver:self
            selector:methodName
                name:notification
              object:nil];
}

-(void) registerNotifications {
[self setUpNotification:NSWorkspaceDidLaunchApplicationNotification
           withSelector:@selector(applicationDidLaunch:)];
```

```
[self setUpNotification:NSWorkspaceDidTerminateApplicationNotification
          withSelector:@selector(applicationDidTerminate:)];
}

-(id) init {
    if (self =[super init]) {
        [self registerNotifications];
    }
    return self;
}
@end
```

Wenn ich nun *Build & Run* anklicke und iCal starte und wieder beende, sehe ich Folgendes in der Konsole:

```
Launched.
Launched.
Terminated.
```

11.8 Notifikationen absetzen

Nehmen wir an, Sie wollen das Fenster um ein Textfeld erweitern und „Launched“ oder „Terminated“ darin ausgeben. Wie ließe sich das realisieren?

Zuerst fügen Sie dem Controller ein Outlet vom Typ `NSTextField` mit dem Namen `activityDisplay` hinzu:

Notication/HelloApplication6/ActivityController.h

```
#import <Cocoa/Cocoa.h>
@class CurrentApp;

@interface ActivityController : NSObject {
    CurrentApp *currentApp;
►   NSTextField *activityDisplay;
}
@property CurrentApp *currentApp;
► @property IBOutlet NSTextField *activityDisplay;
@end
```

Im Interface Builder ziehen Sie ein Multiline-Textfeld in das Anwendungsfenster. Deaktivieren Sie die Checkbox *Selectable* im Attributes Inspector. Verbinden Sie das Outlet mit dem Textfeld und speichern Sie ab.

Wie teilen wir die Start- und Endnotifikationen diesem View mit? Der Controller ist das Objekt, das den View informieren sollte, aber woher bekommt der Controller die Informationen?

Wir könnten den Controller registrieren, sodass er Start- und Endnotifikationen empfängt und an das Modell weiterleitet. Der Controller

kennt das Modell (schließlich hat er es erzeugt), also wäre das nur natürlich. Das Modell könnte den Controller kontaktieren, wenn es Änderungen zu vermelden gibt. Das ist weniger natürlich, weil das Modell noch nichts über den Controller weiß. Im nächsten Kapitel richten wir den Controller als Delegate des Modells ein, und sehr viel später in diesem Buch werden Sie lernen, wie man den Key Value Observer-Mechanismus nutzt.

Eine andere Lösung besteht darin, selbst Notifikationen abzusetzen. Hier setzen wir die Notifikationen `Launched` und `Terminated` mithilfe der Methoden `applicationDidLaunch:` und `applicationDidTerminate:` ab:

Notication/HelloApplication6/CurrentApp.m

```
-(void) applicationDidLaunch: (NSNotification *) notification {
    [[NSNotificationCenter defaultCenter]
                         postNotificationName:@"Launched" object:self];
}
-(void) applicationDidTerminate: (NSNotification *) notification {
    [[NSNotificationCenter defaultCenter]
                        postNotificationName:@"Terminated" object:self];

}
```

Zuerst ermitteln wir das Notification-Center der Anwendung:

```
[NSNotificationCenter defaultCenter]
```

Beachten Sie, dass das nicht dem Aufruf entspricht, den wir verwenden, um ein Handle auf das Notification-Center unserer Workspaces zu ermitteln. Sobald wir das Notification-Center kennen, können wir mit `postNoticationName:object:` eigene Notifikationen absetzen. Wir müssen uns für diese Notifikationen nur noch registrieren und sie verarbeiten.

11.9 Übung: Eigene Notifikationen empfangen

Registrieren Sie im `ActivityController` Observer für die von uns definierten Notifikationen. Implementieren Sie die Methoden `applicationDidLaunch:` und `applicationDidTerminate:` im `ActivityController`, die „Launched“ oder „Terminated“ im Textfeld ausgeben, wenn die entsprechende Notifikation empfangen wird.

11.10 Lösung: Eigene Notifikationen empfangen

Sie kennen die einzelnen Schritte dieser Übung. Fassen wir also zusammen: die Methoden `registerNotications` und `setUpNotication:withSelector:` aus `CurrentApp` einfügen und `registerNotications` aus `awakeFromNib` heraus aufrufen.

Wenn Sie eine Notifikation empfangen, müssen Sie den Stringwert von `activeDisplay` auf „Launched“ oder „Terminated“ setzen.

Notication/HelloApplication6/ActivityController.m

```
#import "ActivityController.h"
#import "CurrentApp.h"

@implementation ActivityController

@synthesize currentApp, activityDisplay;

-(void) applicationDidLaunch: (NSNotification *) notification {
    [self.activityDisplay setStringValue:@"Launched."];
}
-(void) applicationDidTerminate: (NSNotification *) notification {
    [self.activityDisplay setStringValue:@"Terminated."];
}
-(void)setUpNotification:(NSString *)notification withSelector:(SEL)methodName {
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:methodName
     name:notification
     object:nil];
}
-(void) registerNotifications {
    [self setUpNotification:@"Launched"
                withSelector:@selector(applicationDidLaunch:)];
    [self setUpNotification:@"Terminated"
                withSelector:@selector(applicationDidTerminate:)];
}
-(void)awakeFromNib {
    self.currentApp = [[CurrentApp alloc] init];
    [self registerNotifications];
}
@end
```

Mehr über Notifikationen erfahren Sie in Apples *Notication Programming Topics for Cocoa* [App08h].

Im nächsten Kapitel wollen wir diese Anwendung umschreiben und Delegates statt Notifikationen verwenden, um zwischen Modell und Controller zu kommunizieren. Wir müssen dazu unser eigenes Protokoll entwickeln und die dazugehörigen Delegate-Methoden selbst implementieren.

Kapitel 12

Protokolle für die Delegation entwickeln

Notikationen sind eine gute Sache, wenn mehr als ein Objekt über eine Änderung informiert werden muss, oder wenn man im Vorfeld nicht weiß, welche Objekte an einer Benachrichtigung interessiert sein könnten. Die im vorigen Kapitel entwickelte Anwendung fängt zum Beispiel Notifikationen ab, die vom `NSWorkspace` gesendet werden. Das ist ein gutes Einsatzgebiet für Notifikationen.

Andererseits ist die Kommunikation zwischen unserem Modell und unserem Controller kein besonders gutes Einsatzgebiet für Notifikationen.[1] Die `CurrentApp`-Methoden `applicationDidLaunch:` und `applicationDidTerminate:` tun wenig mehr, als Methoden mit im Wesentlichen identischen Namen im `ApplicationController`-Objekt aufzurufen. Und das wird im Augenblick noch auf Umwegen erreicht. Die `CurrentApp`-Methoden setzen Notifikationen ab, für deren Überwachung der `ApplicationController` registriert ist. Wir wären besser dran, wenn wir einen Delegate anstelle von Notifikationen verwenden würden.

1 Warum habe ich es Ihnen gezeigt, wenn es kein gutes Einsatzgebiet für Notifikationen ist? Ich habe Notifikationen auf diese Weise genutzt, um Ihnen zeigen zu können, wie man eigene Notifikationen erzeugt. Mein Ziel ist, Ihnen mithilfe dieses Beispiels eine Vielzahl nützlicher Techniken vorzustellen.

Wir haben Protokolle und Delegates bereits verwendet. In diesem Kapitel werden wir ein eigenes Protokoll entwickeln und in die Header-Datei des Controllers einfügen. Wir werden den Controller zum Delegate für unser Modell machen. Dann kommt der schwierige Teil: Das Modell dreht den Spieß um und ruft die Delegate-Methoden auf, wenn sie existieren.

12.1 Übung: Den Delegate erzeugen und festlegen

Erzeugen Sie eine neue Eigenschaft namens `delegate` in der `CurrentApp`-Klasse. Im Augenblick verwenden wir `id` als Typ des Delegates, wir können ihm also jedes Objekt zuweisen.

Entfernen Sie bis auf `awakeFromNib` alle Methoden aus `ActivityController.m`. Wir werden einige später wieder einfügen, aber das wird Ihnen hier helfen, die Details der Deklaration und Implementierung eines Protokolls zu erkennen. Setzen Sie den `CurrentApp-Delegate` in `awakeFromNib` auf `self`.

12.2 Lösung: Den Delegate erzeugen und festlegen

Fügen Sie eine Instanzvariable und eine Eigenschaft namens `delegate` in `CurrentApp.h` ein:

Protocols/HelloApplication7/CurrentApp.h

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

@interface CurrentApp : NSObject {
    id delegate;
}
@property id delegate;

@end
```

Wir haben den Delegate als `id` gekennzeichnet, sodass ein beliebiges Objekt als `CurrentApp`-Delegate angegeben werden kann. Vergessen Sie nicht, die Eigenschaft in `CurrentApp.m` zu synthetisieren.

Im Moment erledigen wir in der `awakeFromNib`-Methode des `ActivityControllers` nur dreierlei: Wir erzeugen eine Instanz von `CurrentApp`, weisen sie unserer Instanzvariablen zu und legen uns selbst als Delegate für dieses Objekt fest.

Protocols/HelloApplication7/ActivityController.m

```
#import "ActivityController.h"
#import "CurrentApp.h"

@implementation ActivityController

@synthesize currentApp, activityDisplay;

-(void)awakeFromNib {
    self.currentApp = [[CurrentApp alloc] init];
    self.currentApp.delegate = self;
}
@end
```

An dieser Stelle überprüfe ich, ob das Ganze funktioniert, indem ich die folgende einfache Methode in `ActivityController.m` einfüge (die ich später wieder lösche):

```
-(void) sayHi {
    NSLog(@"Hi");
}
```

Am Ende von `awakeFromNib` füge ich die folgende Zeile ein:

```
[self.currentApp.delegate sayHi];
```

Ich führe die App aus und sehe „Hi“ im Konsolenfenster. Ich weiß nun, dass alles funktioniert, und kann `sayHi` und seinen Aufruf wieder löschen. Eine alberne Übung, aber ich weiß jetzt sicher, dass ich weitermachen kann.[2]

12.3 Ein Protokoll entwickeln und benutzen

Ein Protokoll ist eine Sammlung von Methodendeklarationen, die allen Klassen zur Verfügung gestellt wird, die von diesen Methoden einige oder alle implementieren wollen. Sie ähnelt einer Header-Datei, die andere frei übernehmen können. Ein Protokoll erlaubt Ihnen, bestimmte Verhaltensweisen in einer Gruppe von Klassen festzuhalten, die nichts voneinander erben müssen und außer der Implementierung desselben Protokolls auch nichts miteinander zu tun haben müssen.

2 Viele Leser werden einwenden, dass ich zuerst einen Test enwickeln sollte, damit ich auf solche Hacks nicht angewiesen bin. Das ist wahr, aber ich habe für Objective-C noch kein Unit-Testing-Framework gefunden, das ich gut genug finde, um es in dieses Buch aufzunehmen. Ich benutze das in Xcode enthaltene OCUnit. Einen entsprechenden Einstieg bietet Apples *Automated Unit Testing with Xcode 3 and Objective-C* [App09a].

Objective-C ist eine Sprache mit einfacher Vererbung: Eine Klasse kann nur eine Klasse erweitern. Häufig will man anderen mitteilen, dass eine Klasse eine bestimmte andere Klasse erweitert, dass sie aber auch auf eine Reihe anderer Verhaltensweisen reagieren kann. Wir haben das in Kapitel 9, *Anpassungen mit Delegates*, auf Seite 153 gesehen, als wir einen Delegate für das Fenster verwendet haben. Der Delegate war ein `WindowHelper`-Objekt, das `NSWindow` erweitert und das `NSWindowDelegate`-Protokoll implementiert hat. Vielleicht erinnern Sie sich daran, dass uns das eine Liste von Methoden zur Verfügung stellte, die wir implementieren konnten, ohne sie in der Header-Datei explizit deklarieren zu müssen.[3]

Standardmäßig müssen alle für ein Protokoll deklarierten Methoden durch die Klassen implementiert werden, die sie adaptieren. Sie können aber die Direktive `@optional` nutzen. Die dort angegebenen Methoden müssen dann in einer Klasse, die das Protokoll adaptiert, nicht implementiert werden. Die Direktive `@required` führt wiederum erforderliche Methoden auf. Im folgenden Codeschnipsel sind `eineErforderlicheMethode` und `nochEineErforderlicheMethode` erforderlich, während `eineOptionaleMethode` optional ist.

```
@protocol IrgendeinProtokoll

-(void) eineErforderlicheMethode;

@optional
-(void) eineOptionaleMethode;

@required
-(void) nochEineErforderlicheMethode;

@end
```

Lassen Sie uns ein Protokoll entwickeln, das wir zur Definition unseres Delegate benutzen werden. Zuerst legen Sie in Xcode eine neue Header-Datei an. Wählen Sie den Ordner `Classes` in *Groups & Files*. Legen Sie eine neue Datei an und wählen Sie die Vorlage `Mac OS X > Cocoa Class > Objective-C protocol`. Nennen Sie die Datei `ActivityMonitorDelegate.h` und speichern Sie sie.

Die erste Zeile der Datei lautet `@protocol`, gefolgt vom Namen des Protokolls, der dem Namen der Datei entspricht.[4] Protokolle sind Listen von Methodendeklarationen. Es gibt keine Importe, keine Klassen und

3 Ein Objective-C-Protokoll ähnelt einem Interface in Java.

4 Protokoll und Datei können verschiedene Namen haben, aber per Konvention verwendet man den gleichen Namen.

keine Instanzvariablen, weshalb wir keine geschweiften Klammern benötigen. Nachfolgend deklarieren wir `applicationdidLaunch:` und `applicationDidTerminate:` in unserem Protokoll. Beachten Sie, dass diese Methoden einen Parameter vom Typ `CurrentApp` verlangen.

Protocols/HelloApplication7/ActivityMonitorDelegate.h

```
@class CurrentApp;

@protocol ActivityMonitorDelegate

@optional
-(void)applicationDidLaunch: (CurrentApp *) app;
-(void)applicationDidTerminate: (CurrentApp *) app;

@end
```

Nun wollen wir dieses Protokoll in der Klasse `ActivityController` verwenden. Wir müssen eine `import`-Anweisung am Anfang von `ActivityController.h` einfügen. Wir müssen außerdem eine Deklaration einfügen, die besagt, dass `ActivityController` dem Protokoll entspricht. Dazu hängen wir `ActivityMonitorDelegate` in spitzen Klammern an den Namen der `ActivityController`-Superklasse `NSObject` an.

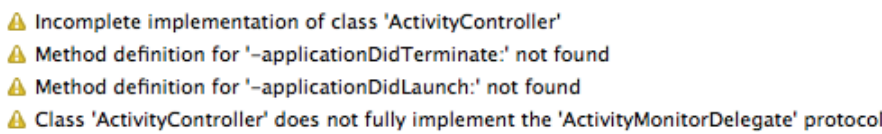
Protocols/HelloApplication7/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"
@class CurrentApp;

@interface ActivityController : NSObject <ActivityMonitorDelegate> {
    CurrentApp *currentApp;
    NSTextField *activityDisplay;
}
@property CurrentApp *currentApp;
@property IBOutlet NSTextField *activityDisplay;
@end
```

Klicken Sie auf *Build & Run*. Es sollte keine Warnungen geben, aber es passiert auch nicht viel, wenn Sie die Anwendung ausführen.

12.4 Methoden verlangen

Entfernen Sie die `@optional`-Zeile aus dem Protokoll und klicken Sie *Build & Run* erneut an. Nun erhalten Sie folgende Warnungen:



Nun verlangt das Protokoll die Methoden `applicationDidLaunch:` und `applicationDidTerminate:`, wir müssen sie also in `ActivityController.m` implementieren.

Protocols/HelloApplication8/ActivityController.m

```
-(void) applicationDidLaunch: (CurrentApp *) app {
    [self.activityDisplay setStringValue:@"Launched"];
}

-(void) applicationDidTerminate: (CurrentApp *) app {
    [self.activityDisplay setStringValue:@"Terminated"];
}
```

Klicken Sie auf *Build & Run*, und die Warnungen sind wieder verschwunden. Wir haben den Delegate eingerichtet und ein Protokoll entwickelt, das die von uns benötigten Methoden deklariert. Wir haben das Protokoll genutzt und die erforderlichen Methoden implementiert. Nun wollen wir die Sache abschließen, indem wir das Absetzen von Notifikationen in `CurrentApp` durch das Senden von Nachrichten an den Delegate ersetzen.

12.5 RespondsToSelector

Sie können wie folgt testen, ob Sie die Delegate-Methode aus dem delegierenden Objekt aufrufen können:

Protocols/HelloApplication8/CurrentApp.m

```
-(void) applicationDidLaunch: (NSNotification *) notification {
    if ([self.delegate
              respondsToSelector:@selector(applicationDidLaunch:)] ) {
         NSLog(@"Delegate implements applicationDidLaunch:.");
    } else {
         NSLog(@"We're on our own.");
    }

}
```

Sie fragen den Delegate, ob er die Methode implementiert, bevor Sie sie aufrufen. Dazu senden Sie die `respondsToSelector:`-Nachricht an den Delegate und übergeben dabei den Namen der Methode, die Sie überprüfen wollen.

Klicken Sie auf *Build & Run*. Die Meldung „Delegate implements applicationDidLaunch:“ sollte im Konsolenfenster erscheinen.

Nun wollen wir ein, zwei Probleme verursachen. Kommentieren Sie die folgende Zeile in `ActivityController.m` aus:

```
self.currentApp.delegate = self;
```

Klicken Sie *Build & Run* an. Diesmal sollte „We're on our own" in der Konsole erscheinen.

Entfernen Sie den Kommentar aus obiger Zeile und kommentieren Sie dafür die Methode `applicationDidLaunch:` in `ActivityController.m` aus. Klicken Sie auf *Build & Run*. Ignorieren Sie die Warnung. Wieder erscheint „We're on our own" in der Konsole.

Aktivieren Sie die Methode wieder, klicken Sie *Build & Run* erneut an und überzeugen Sie sich, dass die Delegate-Methode erfolgreich aufgerufen werden kann.

12.6 Übung: Delegate-Methoden aufrufen

Wir sind bereit, den Delegate fertigzustellen. In der `applicationDidLaunch:`-Methode von `CurrentApp` überprüfen wir, ob der `delegate` auf eine Methode namens `applicationDidLaunch:` reagiert. Ist das der Fall, rufen wir die Methode auf und übergeben dabei `self` als `CurrentApp`-Objekt.

Analog gehen wir auch mit der `applicationDidTerminate:`-Methode von `CurrentApp` vor. Wenn Sie *Build & Run* anklicken, sollten beim Starten und Beenden von Anwendungen „Launched" bzw. „Terminated" im Textfeld erscheinen.

12.7 Lösung: Delegate-Methoden aufrufen

Das ist mal wieder eine von diesen Übungen, deren Beschreibung mehr Zeit erfordert als die eigentliche Lösung. Hier die Änderungen an `applicationDidLaunch:` und `applicationDidTerminate:`:

Protocols/HelloApplication9/CurrentApp.m

```
-(void) applicationDidLaunch: (NSNotification *) notification {
    if ([self.delegate
            respondsToSelector:@selector(applicationDidLaunch:)] ) {
        [self.delegate applicationDidLaunch:self];
    }
}
-(void) applicationDidTerminate: (NSNotification *) notification {
    if ([self.delegate
           respondsToSelector:@selector(applicationDidTerminate:)]) {
        [self.delegate applicationDidTerminate:self];
    }
}
```

Sie müssen außerdem `ActivityMonitorDelegate.h` in die `CurrentApp`-Header-Datei importieren. Sie hätten auch eine Vorwärtsdeklaration in der Header-Datei und einen Import in der Implementierungsdatei benutzen können, aber da ich weiß, dass wir sie gleich sowieso verschieben müssen, habe ich den Import in der Header-Datei platziert.

12.8 Übung: Aufräumen

Hier eine einfache Faustregel für Sie: Wenn Sie etwas im Interface Builder erledigen *können*, dann *sollten* Sie es auch im Interface Builder erledigen.

Sicher, es gibt Ausnahmen, und wir werden manches experimentell mit Code erledigen, das man im IB lösen kann – aber Sie kommen wesentlich weiter, wenn Sie diesem Grundsatz folgen.

Lassen Sie uns das auf unser aktuelles Beispiel anwenden, um ein wenig aufzuräumen. Erzeugen Sie Ihr `CurrentApp`-Objekt im Nib und legen Sie dort auch seinen Delegate fest. Sie müssen an diesem Objekt festhalten, weshalb wir ein Outlet im App-Delegate setzen und es mit dem `CurrentApp`-Objekt verknüpfen.

12.9 Lösung: Aufräumen

Wir können auf die `awakeFromNib`-Methode in `ActivityController.m` verzichten, weil wir das Objekt im Nib erzeugen und dort auch seinen Delegate setzen.

Protocols/HelloApplication10/ActivityController.m

```
#import "ActivityController.h"
#import "CurrentApp.h"

@implementation ActivityController

@synthesize activityDisplay;

-(void) applicationDidLaunch: (CurrentApp *) app {
    [self.activityDisplay setStringValue:@"Launched"];
}
-(void) applicationDidTerminate: (CurrentApp *) app {
    [self.activityDisplay setStringValue:@"Terminated"];
}
@end
```

Wir benötigen keine currentApp-Eigenschaft mehr. Entfernen Sie die Variablen- und Eigenschaftsdeklaration zusammen mit der Vorwärtsdeklaration aus der Header-Datei:

Protocols/HelloApplication10/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface ActivityController : NSObject <ActivityMonitorDelegate> {
    NSTextField *activityDisplay;
}
@property IBOutlet NSTextField *activityDisplay;
@end
```

Wir fügen ein Outlet in den App-Delegate ein, sodass die CurrentApp-Instanz erhalten bleibt, solange wir sie benötigen:

Protocols/HelloApplication10/HelloApplicationAppDelegate.h

```
#import <Cocoa/Cocoa.h>
@class ActivityController;
@class CurrentApp;

@interface HelloApplicationAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
    ActivityController *ac;
    CurrentApp *currentApp;
}
@property IBOutlet NSWindow *window;
@property IBOutlet ActivityController *ac;
@property IBOutlet CurrentApp *currentApp;
@end
```

Denken Sie daran, die currentApp-Eigenschaft in der Implementierungsdatei zu synthetisieren.

Protocols/HelloApplication10/HelloApplicationAppDelegate.m

```
#import "HelloApplicationAppDelegate.h"

@implementation HelloApplicationAppDelegate

@synthesize window, ac, currentApp;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
}
@end
```

Da wir den Delegate nun im IB festlegen, müssen wir den Typ sorgfältiger auswählen. Wir modifizieren CurrentApp.h so:

Protocols/HelloApplication10/CurrentApp.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface CurrentApp : NSObject {
    NSObject <ActivityMonitorDelegate> *delegate;
}
@property IBOutlet NSObject <ActivityMonitorDelegate> *delegate;

@end
```

Wir haben aus `id` ein `NSObject` gemacht. Außerdem habe ich den Protokollnamen in spitzen Klammern eingefügt. Beachten Sie, dass das * zur Variablen gehört. Die Kompilierung schlägt fehl, wenn Sie es an `NSObject` binden. Wir haben außerdem die `delegate`-Eigenschaft als Outlet gekennzeichnet.

Die Änderung der Typdeklaration für die `delegate`-Eigenschaft hat eine angenehme Nebenwirkung: Wir können nun diese Schlussklausel aus der `applicationDidLaunch:`-Methode in `CurrentApp.m` entfernen.

```
if ([self.delegate
          respondsToSelector:@selector(applicationDidLaunch:)] )
```

Wir benötigen diese Kontrolle nicht mehr, nachdem wir deklariert haben, dass der `delegate` das `ActivityMonitorDelegate`-Protokoll implementiert, das die `applicationDidLaunch:`-Methode verlangt. Das Gleiche gilt für die `applicationDidTerminate:`-Methode. Hier sehen Sie die vereinfachten Methoden in `CurrentApp.m`:

Protocols/HelloApplication10/CurrentApp.m

```
-(void) applicationDidLaunch: (NSNotification *) notification {
    [self.delegate applicationDidLaunch:self];
}
-(void) applicationDidTerminate: (NSNotification *) notification {
    [self.delegate applicationDidTerminate:self];
}
```

Speichern Sie ab und wechseln Sie in den Interface Builder. Finden Sie `CurrentApp` im *Classes*-Reiter der Library und ziehen Sie sie in das Dokumentenfenster. Öffnen Sie den Connections Inspector. Wählen Sie das `HelloApplicationAppDelegate`-Objekt und verbinden Sie sein `currentApp`-Outlet mit dem `CurrentApp`-Objekt. Wählen Sie das `CurrentApp`-Objekt und verbinden Sie sein `delegate`-Outlet mit dem `ActivityController`-Objekt. Speichern Sie Ihre Arbeit und beenden Sie den IB.

Klicken Sie auf *Build & Run*, und Ihre Anwendung läuft genau wie vorher.

Wir empfangen nun Notifikationen, wenn eine Anwendung startet und endet, und wir kommunizieren zwischen Modell und Controller über ein Protokoll und einen Delegate. Wir haben mit den empfangenen Notifikationen allerdings noch nicht viel gemacht. Sie enthalten sehr viele Informationen, die wir bisher einfach ignoriert haben. Das wollen wir im nächsten Kapitel ändern.

Kapitel 13

Mit Dictionaries arbeiten

In den letzten Kapiteln haben wir Notifikationen und Delegates verwendet, um „Launched“ und „Terminated“ auszugeben. In diesem Kapitel erfahren Sie, wie man mit Dictionaries arbeitet, um solche Fragen beantworten zu können wie „*Was* wurde gestartet?“ und „*Was* wurde beendet?”.

In den Notifikationen sind sehr viele Informationen enthalten, die wir bisher noch nicht nutzen. Wie bereits erklärt, besitzt `NSNotication` den Namen der Notifikation, das Objekt, das die Notifikation sendet, und ein Dictionary namens `userInfo`, das voll steckt mit Schlüssel/Wert-Paaren mit Informationen zu dieser Notifikation.

In diesem Kapitel tauchen wir in den Inhalt dieses Dictionary ein.[1] Diese Technik zu beherrschen ist wichtig, da Cocoa-Frameworks Dictionaries überall einsetzen.

13.1 Ein Blick auf userinfo

Es gibt drei grundlegende Nutzungsmuster für Notifikationen:

- Im ersten Fall geht es nur darum, dass man eine Notifikation empfängt – Sie wollen nur wissen, ob das Ereignis, auf das Sie warten, eingetreten ist.

1 Je nachdem, welche Programmiersprache Sie sonst benutzen, verwenden Sie statt *Dictionary* vielleicht den Begriff *Hash*.

- Im zweiten Fall könnten Sie den `namen` oder das `object` der Notifikation nutzen wollen.
- Im dritten Fall benötigen Sie weitere Informationen, die im `userInfo`-Dictionary enthalten sind. Wenn Sie beispielsweise eine Notifikation erhalten, dass eine Anwendung gestartet wurde, werden Sie den Namen der Anwendung und ihre Lage auf der Festplatte wissen wollen. Diese Information ist in Form von Schlüssel/Wert-Paaren in einem `NSDictionary` enthalten, das von `userInfo` zurückgegeben wird.

Fügen Sie die folgende Zeile in die `applicationDidLaunch:`-Methode in `CurrentApp.m` ein:

Dictionaries/HelloApplication11/CurrentApp.m

```
-(void) applicationDidLaunch: (NSNotification *) notification {
    self.delegate applicationDidLaunch:self];
►   NSLog(@"%@", notification.userInfo);
}
```

Klicken Sie *Build & Run* an. Sie sollten das `userInfo`-Dictionary im Konsolenfenster sehen, wenn Anwendungen gestartet werden. Wenn ich zum Beispiel iCal starte, sehe ich Folgendes in der Konsole:

```
{
    NSApplicationBundleIdentifier = "com.apple.iCal";
    NSApplicationName = iCal;
    NSApplicationPath = "/Applications/iCal.app";
    NSApplicationProcessIdentifier = 7130;
    NSApplicationProcessSerialNumberHigh = 0;
    NSApplicationProcessSerialNumberLow = 987377;
    NSWorkspaceApplicationKey = "<NSRunningApplication:
                                    0x200019ae0 (com.apple.iCal -7130)>";
}
```

Sie sehen den Namen der Anwendung und ihre Lage auf der Festplatte. Sie sehen (neben anderen Identifiern) den Bundle-Identifier. Wir beginnen damit, den Namen der Anwendung abzurufen.

13.2 Aus einem Dictionary lesen

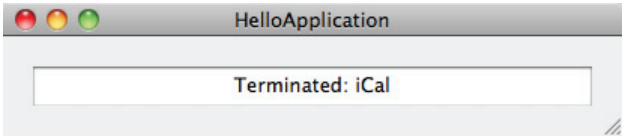
Es gibt viele Möglichkeiten, um Daten aus einem Dictionary auszulesen. Sie können einen Enumerator für den Schlüssel verwenden, um die Dictionary-Einträge durchzugehen und die Werte auszulesen. Sie können sich ein Array aller Schlüssel oder Werte zurückgeben lassen. Wir wollen die Methode `objectForKey:` verwenden:

```
[notification.userInfo objectForKey:@"NSApplicationName"]
```

Wenn Sie nun *Build & Run* anklicken, werden die Namen der gestarteten Anwendungen im Konsolenfenster ausgegeben. So einfach ist das Lesen aus einem Dictionary.

13.3 Übung: Den Namen ausgeben

Fügen Sie die Eigenschaft `name` mit dem Typ `NSString` in `CurrentApp` ein. Wenn Sie eine Notifkation empfangen, setzen Sie die `name`-Eigenschaft auf den Wert, den Sie aus dem `userInfo`-Dictionary einlesen. Ihr Anwendungsfenster sollte etwa so aussehen:



Wenn Sie das Textfeld aktualisieren, soll also „Launched: iCal“ bzw. „Terminated: iCal“ erscheinen, wenn Sie iCal starten oder beenden.

13.4 Lösung: Den Namen ausgeben

Fügen Sie die Instanzvariable und Eigenschaft `name` in die Header-Datei ein. Geben Sie das Speichermodell mit `copy` an, da `NSString` dem `NSCopying`-Protokoll entspricht.

Dictionaries/HelloApplication13/CurrentApp.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface CurrentApp : NSObject {
    NSObject <ActivityMonitorDelegate> *delegate;
►   NSString *name;
}
@property IBOutlet NSObject <ActivityMonitorDelegate> *delegate;
►@property(copy) NSString *name;

@end
```

Setzen Sie den Namen in den `applicationDidLaunch:`- und `applicationDidTerminate:`-Methoden in `CurrentApp.m`:

Dictionaries/HelloApplication13/CurrentApp.m

```
-(void) applicationDidLaunch: (NSNotification *) notification {
►    self.name = [notification.userInfo objectForKey:@"NSApplicationName"];
    [self.delegate applicationDidLaunch:self];
}
-(void) applicationDidTerminate: (NSNotification *) notification {
►    self.name = [notification.userInfo objectForKey:@"NSApplicationName"];
    [self.delegate applicationDidTerminate:self];
}
```

Mit `objectForKey:` können wir einen bestimmten Eintrag aus dem Dictionary einlesen. Wir übergeben `NSApplicationName` als Schlüssel und erhalten den Namen der startenden bzw. endenden Anwendung als `NSString` zurück.

Nutzen Sie die Tatsache, dass die `app` als Parameter an `applicationDidLaunch:` und `applicationDidTerminate:` übergeben wird, um in `ActivityController.m` formatierte Strings zu erzeugen, die die Ausgabe verbessern.

Dictionaries/HelloApplication13/ActivityController.m

```
#import "ActivityController.h"
#import "CurrentApp.h"

@implementation ActivityController

@synthesize activityDisplay;

-(void) applicationDidLaunch: (CurrentApp *) app {
►    [self.activityDisplay setStringValue:
►     [NSString stringWithFormat:@"Launched: %@", app.name]];
}
-(void) applicationDidTerminate: (CurrentApp *) app {
►    [self.activityDisplay setStringValue:
►     [NSString stringWithFormat:@"Terminated: %@", app.name]];
}
@end
```

13.5 Die Redundanz reduzieren

Die beiden `applicationDidXxx:`-Implementierungen in `CurrentApp` sind nahezu identisch. In beiden Fällen habe ich die Überprüfung wieder eingefügt, ob der `delegate` auf unseren Selektor reagiert. Ich schütze mich so vor der Übergabe eines nicht implementierten Methodennamens.

```
if ([self.delegate
        respondsToSelector:@selector(applicationDidXxx:)] ) {
    self.name = [notification.userInfo objectForKey:@"NSApplicationName"];
    [self.delegate applicationDidXxx:self];
}
```

Wir müssen nur sicherstellen, das `Xxx` im einen Fall `Launch` und im anderen `Terminate` lautet. Wir könnten die `notication` an eine Hilfsmethode weitergeben und die aufzurufende Methode basierend auf dem `name` der `notication` auswählen:

Dictionaries/HelloApplication14/CurrentApp.m

```
-(void) respondToChange:(NSNotification *) notification {
    SEL methodName;

    if (notification.name == NSWorkspaceDidLaunchApplicationNotification) {
        methodName = @selector(applicationDidLaunch:);
    } else {
        methodName = @selector(applicationDidTerminate:);
    }
    if ([self.delegate respondsToSelector:methodName]) {
        self.name = [notification.userInfo objectForKey:@"NSApplicationName"];
        [self.delegate performSelector:methodName withObject:self];
    }

}
-(void) applicationDidLaunch: (NSNotification *) notification {
    [self respondToChange:notification];
}
-(void) applicationDidTerminate: (NSNotification *) notification {
    [self respondToChange:notification];
}
```

Das ist viel besser.[2] Wir haben das gemeinsame Verhalten in der Methode `respondToChange:` zusammengefasst. Wir legen den Methodennamen basierend auf dem Wert von `notication.name` mit `applicationDidLaunch:` oder `applicationDidTerminate:` fest.

Einen Unterschied bildet die Art und Weise, in der wir die Methode aufrufen, während wir `self` als Parameter übergeben. Als der Methodenname feststand, konnten wir den Aufruf so durchführen:

```
[self.delegate applicationDidLaunch:self]
```

Nun müssen wir `performSelector:withObject:` nutzen:

```
[self.delegate performSelector:methodName withObject:self];
```

2 Ein Korrektor hat angemerkt, dass wir `applicationDidLaunch:` und `applicationDidTerminate:` an dieser Stelle ganz herausnehmen und die Redundanz noch weiter verringern könnten. Unglücklicherweise müssten wir sie ein paar Abschnitte weiter wieder einfügen, weshalb ich sie beibehalten habe.

Ich bin froh, dass wir den doppelten Code entfernen konnten, bin aber mit der `if`-Anweisung nicht ganz glücklich. Kontrollanweisungen wie `if`, `for`, `case` und so weiter sind nicht grundsätzlich schlecht, aber man sollte sich überlegen, ob es nicht einen direkteren Weg durch den Code gibt.

In unserem einfachen Beispiel gibt es viele Möglichkeiten, um das `if` zu entfernen. Ich werde Ihnen eine Technik zeigen, die Dictionaries nutzt.

13.6 Ein Dictionary zur Flusskontrolle nutzen

Sehen wir uns die `if`-Anweisung noch einmal an:

Dictionaries/HelloApplication14/CurrentApp.m

```
if (notification.name == NSWorkspaceDidLaunchApplicationNotification) {
    methodName = @selector(applicationDidLaunch:);
} else {
    methodName = @selector(applicationDidTerminate:);
}
```

Wir wollen ein Dictionary aufbauen, bei dem die Notifikationsnamen die Schlüssel und die Methodennamen die Werte bilden. Wir können die `if`-Anweisung dann wie folgt ersetzen:

Dictionaries/HelloApplication15/CurrentApp.m

```
SEL methodName = NSSelectorFromString(
                    [delegateMethods objectForKey:[notification name]]);
```

Wir müssen keine Entscheidung treffen, sondern ziehen uns den Methodennamen einfach aus dem Dictionary heraus.

Wir müssen eine Instanzvariable namens `delegateMethods` vom Typ `NSDictionary` in `CurrentApp.h` definieren. Sie muss keine Eigenschaft sein. Wir initialisieren Sie so:[3]

Dictionaries/HelloApplication15/CurrentApp.m

```
-(void) initializeMethodDictionary {
    delegateMethods = [[NSDictionary alloc] initWithObjectsAndKeys:
                           @"applicationDidLaunch:",
                           NSWorkspaceDidLaunchApplicationNotification,
                           @"applicationDidTerminate:",
                           NSWorkspaceDidTerminateApplicationNotification,
                           nil];
}
```

3 Fügen Sie einen Aufruf von `initializeMethodDictionary` an das Ende der `init`-Methode ein.

Wir übergeben die Schlüssel/Wert-Paare als kommaseparierte, `nil`-terminierte Liste. Für jedes Paar übergeben wir zuerst den Wert und dann den Schlüssel. Wir speichern die Methodennamen als `NSStrings` ab, weil die in ein Dictionary eingefügten Werte Objekte sein müssen.

13.7 Einträge mit einem mutablen Dictionary einfügen und entfernen

Es gibt Situationen, in denen Sie im Dictionary Einträge entfernen bzw. einfügen wollen. Solche Änderungen können Sie nicht am `NSDictionary` vornehmen, weil es unveränderlich (immutable) ist. Stattdessen müssen Sie `NSMutableDictionary` verwenden, eine Subklasse von `NSDictionary`, die solche Änderungen erlaubt.

Möglicherweise fanden Sie es frustrierend, dass die Apple-Dokumentation einer Klasse nicht alle Methoden aufführt, die für Objekte eines bestimmten Typs zur Verfügung stehen. Die Dokumentation enthält umfassende Beschreibungen der Methoden, die für diese Klasse definiert sind, führt aber keine Methoden auf, die von den Superklassen vererbt werden. Sie müssen der Dokumentation durch die ganze Hierarchie folgen, um eine bestimmte Methode zu finden, die mehrere Ebenen weiter oben definiert ist (häufig in einer Klasse, an die Sie nie gedacht hätten).[4]

Doch diese allgemeine Unbequemlichkeit kann manchmal auch von Vorteil sein. Wenn Sie sich zum Beispiel die Dokumentation für `NSMutableDictionary` ansehen, erkennen Sie sofort, wie sie sich von `NSDictionary` unterscheidet. Nur die zusätzlichen Methoden sind aufgeführt. Es gibt Methoden zur Erzeugung eines mutablen Dictionary, die es Ihnen erlauben, die anfängliche Kapazität des Dictionary festzulegen.[5] Es gibt zusätzliche Methoden zum Hinzufügen und Entfernen von Einträgen. In jeder anderen Hinsicht verhält sich ein `NSMutableDictionary` genau wie ein `NSDictionary`, alle Methoden der Superklasse stehen also auch in `NSMutableDictionary` zur Verfügung.

Lassen Sie uns also ein wenig mit `NSMutableDictionary` herumspielen. Zuerst deklarieren Sie eine Instanzvariable namens `runningApps` in `CurrentApp.h`.

4 Nachdem Sie den Namen des Ziels und ein Leerzeichen eingegeben haben, können Sie immer *Escape* drücken, um sich alle verfügbaren Methoden anzusehen. Nutzen Sie das zusammen mit Quick Help, um schnell zu verstehen, was eine Methode macht.

5 Wenn das Dictionary wächst, wird automatisch zusätzlicher Speicherplatz alloziert.

Dictionaries/HelloApplication16/CurrentApp.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface CurrentApp : NSObject {
    NSObject <ActivityMonitorDelegate> *delegate;
    NSString *name;
    NSDictionary *delegateMethods;
►   NSMutableDictionary *runningApps;
}
@property IBOutlet NSObject <ActivityMonitorDelegate> *delegate;
@property(copy) NSString *name;
@end
```

Wir erzeugen unser mutables Dictionary in der `init`-Methode. Nutzen Sie die Methode `setObject:forKey`, um einen neuen Eintrag in das Dictionary aufzunehmen. Bei jedem Start einer Anwendung verwendet dieser den Namen der Anwendung als Schlüssel und einen Timestamp als Wert.[6]

```
[runningApps setObject:[NSDate date] forKey:appName];
```

Für `appName` müssen wir den Namen der Anwendung aus dem `userInfo`-Dictionary der `notication` bestimmen. In gleicher Weise nutzen wir die Methode `removeObjectForKey:`, wenn eine Anwendung beendet wurde, um den Eintrag mit demjenigen Schlüssel zu entfernen, der dem Namen der beendeten Anwendung entspricht.

```
[runningApps removeObjectForKey: appName];
```

Wir erweitern unser Logging um die Ausgabe des Werts von `runningApps` am Ende der `respondToChange:`-Methode. Wenn ich die Anwendung ausführe und iCal starte, sehe ich Folgendes in meiner Konsole:

```
HelloApplication = "2009-10-03 10:45:36 -0400";
iCal = "2009-10-03 10:45:39 -0400";
```

Hier der Vollständigkeit halber unsere `CurrentApp.m`:

Dictionaries/HelloApplication16/CurrentApp.m

```
#import "CurrentApp.h"

@implementation CurrentApp

@synthesize delegate, name;
```

6 Dafür gibt es eigentlich keinen Grund. Wir wollen nur zeigen, dass wir Einträge in unser Dictionary einfügen und sie wieder löschen können.

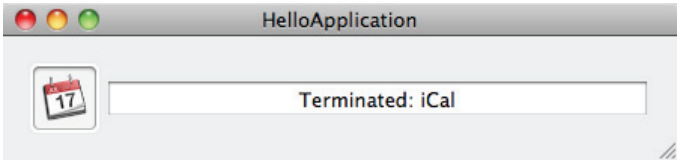
```
-(void) respondToChange:(NSNotification *) notification {
    SEL methodName = NSSelectorFromString(
                    [delegateMethods objectForKey:[notification name]]);
    if ([self.delegate respondsToSelector:methodName]) {
         self.name = [notification.userInfo objectForKey:@"NSApplicationName"];
        [self.delegate performSelector:methodName withObject:self];
    }
    NSLog(@"%@", runningApps);
}
-(void) applicationDidLaunch: (NSNotification *) notification {
    [runningApps setObject:[NSDate date]
                    forKey:[notification.userInfo
                                          objectForKey:@"NSApplicationName"]];
    [self respondToChange:notification];
}
-(void) applicationDidTerminate: (NSNotification *) notification {
    [runningApps removeObjectForKey:[notification.userInfo
                                           objectForKey:@"NSApplicationName"]];
              [self respondToChange:notification];
}
-(void)setUpNotification:(NSString *)notification withSelector:(SEL)methodName {
    [[[NSWorkspace sharedWorkspace] notificationCenter]
               addObserver:self
                  selector:methodName
                      name:notification
                    object:nil];
}
-(void) registerNotifications {
    [self setUpNotification:NSWorkspaceDidLaunchApplicationNotification
                withSelector:@selector(applicationDidLaunch:)];
    [self setUpNotification:NSWorkspaceDidTerminateApplicationNotification
                withSelector:@selector(applicationDidTerminate:)];
}
-(void) initializeMethodDictionary {
    delegateMethods = [[NSDictionary alloc] initWithObjectsAndKeys:
                            @"applicationDidLaunch:",
                            NSWorkspaceDidLaunchApplicationNotification,
                            @"applicationDidTerminate:",
                            NSWorkspaceDidTerminateApplicationNotification,
                            nil];
}
-(id) init {
    if (self =[super init]) {
        [self registerNotifications];
        [self initializeMethodDictionary];
        runningApps = [[NSMutableDictionary alloc] initWithCapacity:5];
    }
    return self;
}
@end
```

Bevor Sie weitermachen, entfernen Sie bitte alle im obigen Code hervorgehobenen Zeilen in den Header- und Implementierungsdateien von `CurrentApp`. Sie richten keinen Schaden an, lassen unser Beispiel aber ein wenig unordentlich wirken. Nachdem wir nun wissen, wie man ein mutables Dictionary erzeugt und damit arbeitet, benötigen wir den Code nicht mehr.

13.8 Übung: Ein Icon hinzufügen

Nachdem wir unseren Code aufgeräumt haben, wollen wir das `userInfo`-Dictionary nutzen, um unsere *HelloApplication* um ein weiteres Feature zu erweitern. Lassen Sie uns das Icon der Anwendung ausgeben, die gerade gestartet oder beendet wird.

Wenn wir also beispielsweise iCal beenden, sollte unsere Ausgabe so aussehen:



Wir können das erreichen, indem wir ein Outlet in die `ActivityController`-Header-Datei einfügen, ein entsprechendes GUI-Element im Interface Builder einbinden oder unseren Code umgestalten. Fangen wir mit den einfachen Schritten an.

In Xcode fügen Sie ein Outlet namens `imageView` in die Header-Datei `ActivityController.h` ein. Dieses Outlet muss ein Zeiger auf einen `NSImageView` sein.

Im Interface Builder wählen Sie einen `NSImageView` in der Library aus, der als *image well* (Bildquelle) beschrieben wird. Passen Sie Fenster, Textfeld und Image so an, dass sie wie im obigen Bild aussehen.

Passen Sie die Größeneinstellungen so an, dass der Benutzer nur die horizontale Größe des Fensters verändern kann. Wenn das Fenster wächst, soll das Icon an derselben Stelle stehen bleiben, während das Textfeld mitwachsen soll, um den Abstand auf der linken und rechten Seite gleich zu halten.

Bevor Sie den Interface Builder verlassen, verbinden Sie das `imageView`-Outlet mit der Bildquelle. Speichern Sie ab.

Wir machen uns eine neue Klasse zunutze, die mit Snow Leopard eingeführt wurde. `NSRunningApplication` erlaubt Ihnen das Senden von Nachrichten und das Abfragen von Informationen zu einer gerade laufenden Anwendung. Sehen Sie sich noch einmal die letzte Ausgabe des `userInfo`-Dictionary im Konsolenfenster an:

```
NSWorkspaceApplicationKey = "<NSRunningApplication:
                                0x200019ae0 (com.apple.iCal -7130)>";
```

Die `NSRunningApplication` hat die folgenden Eigenschaften:

```
activationPolicy
active
bundleIdentifier
bundleURL
executableArchitecture
executableURL
finishedLaunching
hidden
icon
launchDate
localizedName
processIdentifier
terminated
```

All diese Informationen sind im Rest des `userInfo`-Dictionary enthalten; weitere finden Sie in der Instanz von `NSRunningApplication` mit dem Schlüssel `NSWorkspaceApplicationKey`.

Weil wir unter Snow Leopard entwickeln, können Sie ihre Anwendung so umgestalten, dass sie `NSRunningApplication` verwendet, wann immer es möglich ist. Statt den Namen der Anwendung aus dem `userinfo`-Dictionary abzurufen und zu speichern, können wir die laufende Anwendung abrufen und speichern. Und statt eine Instanz von `CurrentApp` an unsere Delegate-Methoden zu übergeben, können wir die laufende Anwendung benutzen, um den Namen und das Icon auszugeben.[7]

Nehmen Sie sich ein paar Minuten Zeit, um selbst eine Lösung zu finden, bevor Sie die nachfolgende Lösung durchgehen.

7 Dieser Mehraufwand war unnötig, als wir nur den Namen der Anwendung benötigen. Jetzt ist es aber sinnvoll, die `NSRunningApplication`-Instanz zu verwenden, die wir aus dem `userInfo`-Dictionary abrufen.

13.9 Ein Icon hinzufügen

Ihre Header-Datei sollte wie folgt aussehen:

Dictionaries/HelloApplication17/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface ActivityController : NSObject <ActivityMonitorDelegate> {
     NSTextField *activityDisplay;
►    NSImageView *imageView;
}
@property IBOutlet NSTextField *activityDisplay;
►@property IBOutlet NSImageView *imageView;
@end
```

Im Interface Builder sollten Sie keine Schwierigkeiten haben, die Bildquelle einzufügen und mithilfe der blauen Hilfslinien die Bildquelle und das Textfeld innerhalb des Fensters anzuordnen. Sie sollten den Size Inspector verwendet haben, um die minimale und maximale Größe des Fensters festzulegen. Wenn die Höhe für die minimale und maximale Größe identisch ist, kann der Benutzer nur die Breite des Fensters anpassen. Sie sollten den Size Inspector auch für das Icon und das Textfeld verwendet haben, damit diese korrekt dargestellt werden, wenn die Größe des Fensters verändert wird.

Genau wie beim Code wollen wir auch hier am Ende anfangen. Wenn wir `NSRunningApplication` als Parameter an die `ActivityController`-Methoden `applicationDidLaunch:` und `applicationDidTerminate:` übergeben, können wir den Namen und das Icon der Anwendung wie folgt bestimmen:[8]

Dictionaries/HelloApplication17/ActivityController.m

```
#import "ActivityController.h"

@implementation ActivityController

@synthesize activityDisplay,imageView;

-(void) applicationDidLaunch: (NSRunningApplication *) app {
   [self.activityDisplay setStringValue:
    [NSString stringWithFormat:@"Launched: %@", app.localizedName]];
   [self.imageView setImage:app.icon];
}
```

8 Es fühlt sich etwas seltsam an, wenn wir eine endende Anwendung als laufende Anwendung bezeichnen, doch es geht um die `NSRunningApplication`-Instanz des `userInfo`-Dictionary, die in der von uns empfangenen Notifikation enthalten ist.

```
-(void) applicationDidTerminate: (NSRunningApplication *) app {
   [self.activityDisplay setStringValue:
    [NSString stringWithFormat:@"Terminated: %@", app.localizedName]];
   [self.imageView setImage:app.icon];
}
@end
```

Wir können Redundanz vermeiden, indem wir die Hilfsmethode `displayAction:forApplication:` einführen:

Dictionaries/HelloApplication18/ActivityController.m

```
#import "ActivityController.h"

@implementation ActivityController

@synthesize activityDisplay,imageView;

-(void) displayAction:(NSString *) action
           forApplication:(NSRunningApplication *) app {
    [self.activityDisplay setStringValue:
        [NSString stringWithFormat:@"%@: %@",action, app.localizedName]];
    [self.imageView setImage:app.icon];
}

-(void) applicationDidLaunch: (NSRunningApplication *) app {
    [self displayAction:@"Launched" forApplication:app];
}
-(void) applicationDidTerminate: (NSRunningApplication *) app {
    [self displayAction:@"Terminated" forApplication:app];
}
@end
```

Die Signatur unserer Delegate-Methoden hat sich geändert, weshalb wir das Protokoll entsprechend korrigieren müssen:

Dictionaries/HelloApplication18/ActivityMonitorDelegate.h

```
@protocol ActivityMonitorDelegate

-(void)applicationDidLaunch: (NSRunningApplication *) app;
-(void)applicationDidTerminate: (NSRunningApplication *) app;

@end
```

Ein weiterer Vorteil des `NSRunningApplication`-Objekts besteht darin, dass wie den `import` und die Vorwärtsdeklarationen aus der `CurrentApp`-Klasse entfernen können.

In `CurrentApp.h` ersetzen wir die Eigenschaft `name` durch eine Eigenschaft namens `app` vom Typ `NSRunningApplication`:

Dictionaries/HelloApplication18/CurrentApp.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface CurrentApp : NSObject {
    NSObject <ActivityMonitorDelegate> *delegate;
►   NSRunningApplication *app;
    NSDictionary *delegateMethods;
}
@property IBOutlet NSObject <ActivityMonitorDelegate> *delegate;
► @property NSRunningApplication *app;

@end
```

Abschließend synthetisieren wir die app-Eigenschaft, setzen ihren Wert über das userInfo-Dictionary und übergeben sie als Parameter beim Aufruf der Delegate-Methoden:

Dictionaries/HelloApplication18/CurrentApp.m

```
► @synthesize delegate, app;

-(void) respondToChange:(NSNotification *) notification {
    SEL methodName = NSSelectorFromString(
              [delegateMethods objectForKey:[notification name]]);

    if ([self.delegate respondsToSelector:methodName]) {
►       self.app = [notification.userInfo
►           objectForKey:@"NSWorkspaceApplicationKey"];
►       [self.delegate performSelector:methodName withObject:self.app];
    }
}
```

Sie werden gewisse Ähnlichkeiten feststellen, wenn Sie sehen, wie Objekte über ihre Schlüssel aus Dictionaries ermittelt werden, und wie Sie auf die Eigenschaften eines Objekts zugreifen. Wir werden uns diesem Thema später noch widmen.

Sie können nun ein Dictionary verwenden, um Informationen aus einer Notifikation abzurufen, die Sie jedesmal erhalten, wenn eine Anwendung gestartet und beendet wird. Sie haben diese Informationen benutzt, um Text und Images an Standard-GUI-Widgets zu senden. Im nächsten Kapitel werden Sie sehen, wie man einen eigenen View entwickelt, um diese Informationen etwas flexibler darzustellen.

Kapitel 14

Mehrere Nibs

Bisher befindet sich bei unserer Desktopanwendung alles in einem einzigen Nib. In diesem Kapitel wollen wir das Nib zuerst in zwei und dann in drei Teile zerlegen. Auch wenn wir uns keine großen Gedanken darum gemacht haben, wurden bei der Entwicklung unseres iPhone-Webbrowsers zwei *.nib*-Dateien verwendet. Eine enthielt unser Fenster und den View-Controller und die andere enthielt den vom View-Controller kontrollierten View.

In diesem Kapitel will ich Ihnen zeigen, wie man mit mehreren *.nib*-Dateien arbeitet. Genau wie Sie Methoden oder Klassen aufteilen, wenn sie zu groß werden, lernen Sie hier, wie man Nibs in kleine, einfach zu beschreibende Sammlungen von Objekten zerlegt. Genau wie bei Objekten und Methoden, die man in kleinere Teile zerlegt, werden Sie häufig mit den einzelnen Teilen kommunizieren müssen. Das ist die Aufgabe des *File's Owner*. Am Ende dieses Kapitels werden Sie auch verstehen, wie der *File's Owner* funktioniert.

14.1 Methoden, Objekte und Nibs

Ein Nib bildet eine der Organisationsebenen Ihres Projekts. Ein Nib ist eine Sammlung von Objekten und ihren Verbindungen untereinander.

Näher betrachtet, ist jedes Objekt eine zusammenhängende Sammlung von Funktionalitäten. Ein Objekt kann Variablen, Eigenschaften, Funktionen und Methoden umfassen. Das bedeutet natürlich, dass Objekte eine Kombination aus Zuständen und Verhalten sind, aber ich denke bei Objekten lieber an das, was sie tun. Ich will damit sagen, dass Objekte für mich durch ihre Methoden charakterisiert werden.

Methoden bilden die kleinste Organisationsebene, die ich mir im Moment vorstellen möchte. Wir entwickeln eine Methode, wenn wir eine Aufgabe erledigen, eine Aktion implementieren oder eine Berechnung durchführen wollen. Zusammengehörende Methoden werden in einer Klasse zusammengefasst. Objekte sind Instanzen dieser Klassen. Objekte, die miteinander kommunizieren müssen, fassen wir häufig in einer *.nib*-Datei zusammen.

Mit anderen Worten stellt ein Nib eine zusammengehörende Sammlung von Objekten, ihren Anfangszuständen und den Verbindungen untereinander dar, und zwar in der gleichen Weise, wie ein Objekt eine Sammlung von Variablen, ihren Anfangszuständen und den vom Objekt verstandenen Nachrichten ist. In den nächsten Kapiteln werden Sie sehen, dass ein Nib für Ihre Anwendung nur wenig mehr ist als ein einzelnes Objekt.

Bevor wir damit beginnen, das Nib in kleinere Teile zu zerlegen, wollen wir noch einmal über die Vor- und Nachteile nachdenken, die die Refaktorierung von Methoden oder Klassen mit sich bringt. Kürzere Methoden lassen sich leichter beschreiben, und man kann einfacher verstehen, was sie genau machen. Wenn wir zum Beispiel eine neue Instanz von `CurrentApp` erzeugen, machen wir momentan Folgendes:

Dictionaries/HelloApplication18/CurrentApp.m

```
-(id) init {
    if (self =[super init]) {
        [self registerNotifications];
        [self initializeMethodDictionary];
    }
    return self;
}
```

Wir registrieren die Notifikationen für dieses Objekt und initialisieren das Methoden-Dictionary. Diese Methode liest sich wie Prosa. Die `registerNotications`-Methode sieht so aus:

Dictionaries/HelloApplication18/CurrentApp.m

```
-(void) registerNotifications {
    [self setUpNotification:NSWorkspaceDidLaunchApplicationNotification
            withSelector:@selector(applicationDidLaunch:)];
    [self setUpNotification:NSWorkspaceDidTerminateApplicationNotification
            withSelector:@selector(applicationDidTerminate:)];
}
```

Auch das kann man, wenn man sich an die Objective-C-Syntax gewöhnt hat, ziemlich leicht lesen. Wir verknüpfen die „gestartet“-Notifikation mit der Callback-Methode `applicationDidLaunch:`, und die Notifikation „beendet“ mit der Callback-Methode `applicationDidTerminate:`. Wie das genau geschieht, ist in der Methode `setUpNotication:withSelector:` festgelegt.

Dictionaries/HelloApplication18/CurrentApp.m

```
-(void)setUpNotification:(NSString *)notification withSelector:(SEL)methodName {
    [[[NSWorkspace sharedWorkspace] notificationCenter]
                addObserver:self
                   selector:methodName
                       name:notification
                     object:nil];
}
```

Diese Methode ist einfach ein Wrapper, in dem wir unsere Notifikationen im Notification Center des gemeinsamen Workspace registrieren.

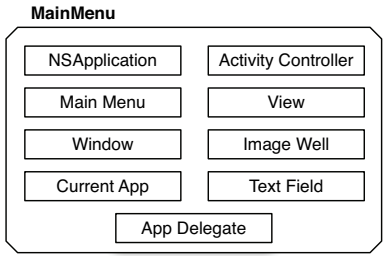
Den Prozess in kleinere Teile zu zerlegen, vereinfacht das Verständnis der einzelnen Teile. Wir verstehen sofort, *was* in der `init`-Methode geschieht, auch wenn wir nicht im Detail wissen, *wie* es geschieht. Wir hätten eine lange `init`-Methode entwickeln können, die jeden Aufruf durch den tatsächlichen Code ersetzt. Wir hätten uns keine Namen für die Methoden überlegen und Informationen übergeben müssen. Alles wäre an einem Ort versammelt.

Das Gleiche gilt für den Entwurf von Klassen und Objekten. Wir haben das Verhalten des Modells in `CurrentApp` isoliert und das Verhalten des Controllers in `ActivityController`. Jeder Teil für sich ist nun einfacher zu verstehen, doch wir haben die letzten drei Kapitel damit verbracht, unterschiedliche Formen der Kommunikation zwischen ihnen zu erklären. Wir hätten die Implementierung vielleicht vereinfachen können, indem wir das Ganze in ein einzelnes, schwer zu verstehendes Objekt gepackt hätten.

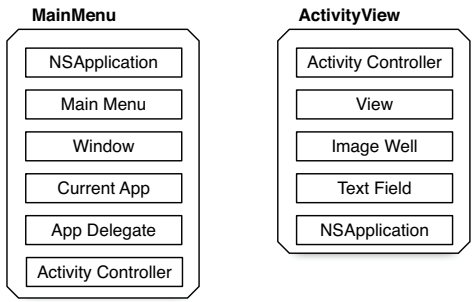
In diesem Fall habe ich sie getrennt, weil ich auf diese Weise eine Flexibilität erhalte, die ich anders nicht erreichen kann. Ich bin in der Lage, das gleiche Modell für einen völlig anderen Controller und View zu verwenden, ohne eine Zeile des Codes in `CurrentApp.m` neu schreiben zu müssen. Wenn ich später Korrekturen am Code vornehmen muss, weiß ich, ob diese Änderung im Modell oder im Controller erfolgen muss. Mit anderen Worten ist in diesem Fall die Einfachheit der Objekte wichtiger als die Mühen bei der Kommunikation zwischen ihnen.

14.2 Nibs aufteilen

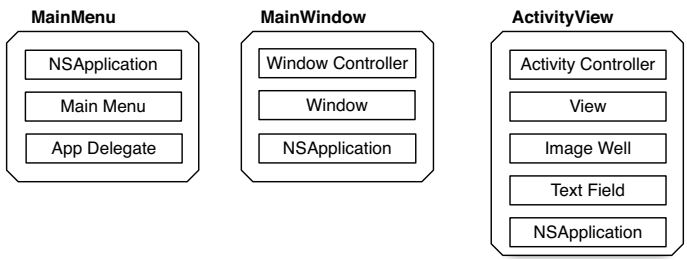
Wir besitzen eine einzelne *.nib*-Datei namens `MainMenu.xib`, die eine Vielzahl von Objekten enthält.



Der `ActivityController` stellt die Brücke zwischen zwei Welten dar: Er muss mit den Komponenten des Views kommunizieren, um das Icon und die Nachricht darzustellen. Wir haben diese Komponenten im Nib platziert. Das `ActivityController`-Objekt muss außerdem mit dem Modell kommunizieren. Wir werden das Nib in diese zwei Teile zerlegen, wobei in jedem Teil die gleiche Instanz des `ActivityController` vorhanden ist:



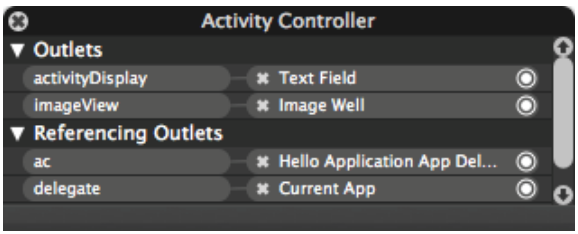
Der `ActivityView` ist eine zusammengehörende Sammlung von Objekten. Er besteht aus den Komponenten des Views und dem View-Controller. `MainMenu.xib` kann noch weiter zerlegt werden, indem man `NSWindow` in ein eigenes Nib auslagert. Wir müssen dazu einen Window-Controller einführen, der unser Fenster in der gleichen Weise verwaltet, wie das der View-Controller für unseren View tut.



Das zu Beginn jedes Nib positionierte Objekt ist der sogenannte *File's Owner*. Wie Sie gleich sehen werden, ist der *File's Owner* das Objekt, dass das neue Nib mit dem Rest der Anwendung verknüpft. Wir teilen also ein einzelnes Nib in drei kleinere Teile auf, die über deren *File's Owners* verknüpft werden. Lassen Sie uns damit beginnen, das erste Nib herauszulösen.

14.3 Die Ausgliederung des Views vorbereiten

Beginnen wollen wir mit der Ausgliederung des Views. Den Schlüssel bildet hierbei der View-Controller. Wir werden das vom View-Controller kontrollierte Textfeld und die Bildquelle in ein neues Nib verschieben, das einen View enthält. Wir fügen diesen View in das Fenster ein, verschieben aber weder Fenster noch Modell oder App-Delegate. Hier die aktuellen Verbindungen des `ActivityController`:



Beachten Sie, dass unser `ActivityController` mit vier Objekten verknüpft ist, die bald in zwei verschiedenen Nibs enthalten sein werden. Der Trick besteht darin, sicherzustellen, dass dieselbe Instanz des `ActivityController` mit beiden Nibs kommunizieren kann. Wir werden einen `ActivityController` im `MainMenu`-Nib anlegen und als *File's Owner* in unserem neuen Nib nutzen. Ein Ziel dieser Übung besteht darin, Ihnen die Rolle des *File's Owner* zu verdeutlichen.

Wir haben den `ActivityController` als View-Controller verwendet, doch bislang hat er die `NSViewController`-Klasse nicht erweitert. Wir korrigieren das, indem wir die Superklasse in der `ActivityController`-Header-Datei ändern:

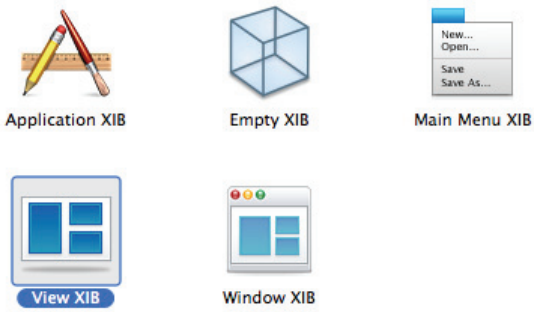
MultipleNibs/HelloApplication19/ActivityController.h

```
@interface ActivityController : NSViewController <ActivityMonitorDelegate> {
```

Wenn Sie sich nun die Outlets des `ActivityController` im Connections Inspector anschauen, werden Sie ein neues `view`-Outlet sehen. Wir werden es im nächsten Abschnitt verwenden, wenn wir aus einem Nib zwei machen.

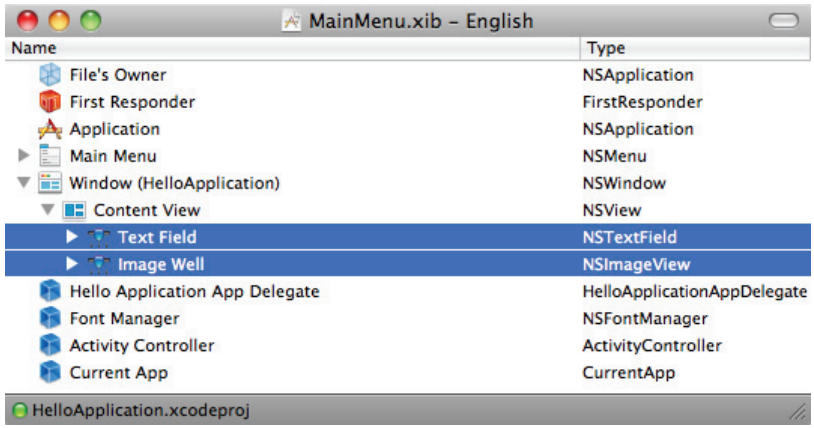
14.4 Das View-Nib anlegen

Als Nächstes legen wir die neue *.nib*-Datei an, die unser Textfeld und unsere Bildquelle enthalten wird. Wählen Sie den Ordner *Resources* in *Groups & Files* und legen Sie eine neue Datei an. Diesmal wählen Sie `Mac OS X > User Interface > View XIB`.

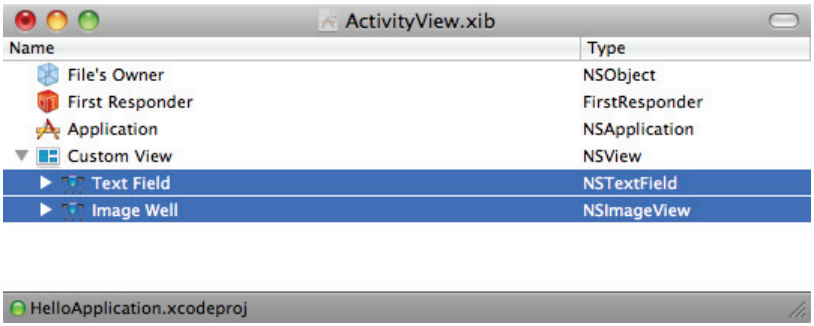


Nennen Sie die *.nib*-Datei `ActivityView.xib` und klicken Sie auf den *Finish*-Button.

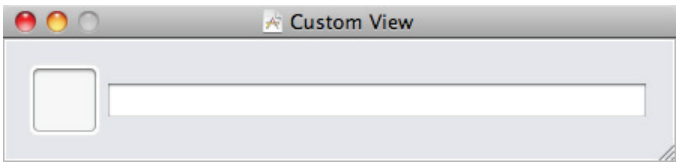
Nun sind einige chirurgische Eingriffe notwendig. Wir wollen zwei Elemente aus dem `MainMenu`-Nib in das `ActivityView`-Nib verschieben. Sehen Sie sich das Dokumentenfenster in `MainMenu.xib` an und lokalisieren Sie das Textfeld und die Bildquelle.



Ziehen Sie diese beiden Elemente aus `MainMenu.xib` nach `ActivityView.xib` und positionieren Sie sie so, dass Sie Child-Elemente des Custom View sind.



Entfernen Sie sie aus `MainMenu.xib`. Wechseln Sie zu `ActivityView.xib` und klicken Sie den Custom View doppelt an. Positionieren Sie Bildquelle und Textfeld neu und passen Sie die Größe des Views an.



Es sind nur noch wenige Schritte nötig, und alles ist wieder verbunden. Zuerst wollen wir Nib und *File's Owner* besser verstehen.

14.5 Eine .nib-Datei integrieren

Machen Sie sich noch einmal bewusst, dass das Nib ein schockgefrorener Graph von Objekten ist. Für alle *.nib*-Dateien, die Sie entwickeln und zum Leben erwecken, benötigen Sie ein Objekt, das die Nachricht zum Laden des Nib sendet. Es muss auch ein Objekt geben, das die Verbindungen der Objekte in der *.nib*-Datei mit den Objekten in der bereits laufenden Anwendung herstellt. Für diese erste Rolle gibt es keinen formellen Namen. Das Objekt, das für die Verbindung des Nib mit der laufenden Anwendung verantwortlich ist, wird *File's Owner* (also „Besitzer der Datei“) genannt.

Jedes Nib benötigt ein Objekt namens *File's Owner*. Dieses Objekt muss existieren, bevor das Nib geladen wird. Wie Sie in den Beispielen dieses Kapitels sehen werden, kann der *File's Owner* sogar das Objekt sein, dass das Nib lädt. Beim `MainMenu`-Nib ist das Anwendungsobjekt der *File's Owner*. Es wird instanziiert, bevor das Nib geladen wird. An irgendeinem Punkt wird unser neues `ActivityView`-Nib geladen. Wir legen dessen *File's Owner* mit dem `ActivityController`-Objekt fest, das im `MainMenu`-Nib erzeugt wurde.

Um die Rolle des *File's Owner* zu verstehen, stellen Sie sich eine Party vor, an der Sie und ein paar Freunde teilnehmen wollen. Sie wissen nicht, wo die Party ist, und kennen die anderen Gäste nicht. Eine Stimme am Telefon fordert Sie auf, geduldig zu sein und zu warten, bis Sie jemand abholt. Sie werden zu dieser Party gebracht und Ihrer *Verbindung* vorgestellt.

„Wie werde ich diese ‚Verbindung' erkennen?", werden Sie fragen.

„Sie werden den Typ erkennen", wird man Ihnen sagen.

Und so hängen Sie und Ihre Freunde rum, genau wie der in einer *.nib*-Datei eingefrorene Objektgraph. Während Sie so im Nib abhängen, ist die Anwendung wie die Party, wo es im Moment noch ohne Sie abgeht. Stunden vergehen. Auf der Party interagieren Objekte und senden sich Nachrichten hin und her.

Da klopft es an der Tür. Sie und Ihre Freunde recken und strecken sich und fangen an zu plaudern. Die Tür wird geöffnet.

"Vorwärts", sagt eine barsche Stimme, „alle ab in den Wagen!" Ah, das ist das Objekt, dessen Aufgabe darin besteht, das Nib zu laden.

Sie erreichen die Party und steigen aus dem Wagen. Sie und Ihre Freunde sehen sich um und fühlen sich verloren. Sie sehen den Fahrer fragend an. Er zeigt auf einen schwarz gekleideten Mann. Das ist der Typ. Sie haben ihn noch nie gesehen, aber die Stimme am Telefon hatte recht: Sie *erkennen* den Typ. Das ist Ihre Verbindung – Ihr *File's Owner*. Er führt Sie und Ihre Gruppe ein und stellt sie vor.

Alternatives Ende

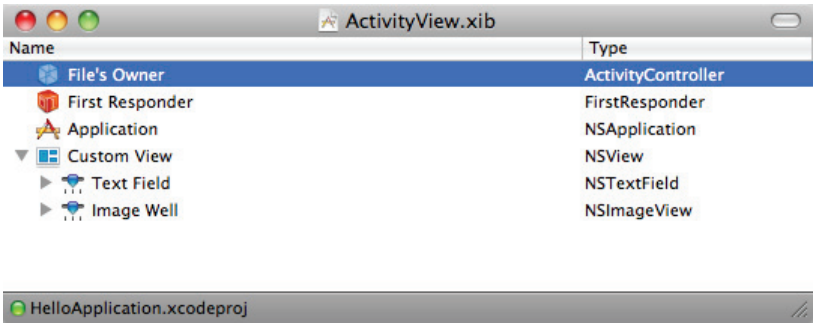
Sie erreichen die Party und steigen aus dem Wagen. Sie und Ihre Freunde sehen sich um und fühlen sich verloren. Sie sehen den Fahrer fragend an. Er grinst verschmitzt zurück und Sie erkennen, dass *er* der Typ ist. Die Stimme am Telefon hatte recht. Wenn Sie genauer hinsehen, *erkennen* Sie den Typ. Das ist Ihre Verbindung – Ihr *File's Owner*. Er führt Sie und Ihre Gruppe ein uns stellt sie vor.

Warum zwei Enden? Im ersten Fall ist der *File's Owner* nicht das Objekt, das das Nib lädt. Im zweiten Fall handelt es sich um dasselbe Objekt. In beiden Fällen weiß das Nib, welchen Typ es erwartet, doch es ist Aufgabe des Objekts, das das Nib lädt, dasjenige Objekt dieses Typs anzugeben, das tatsächlich *File's Owner* ist.

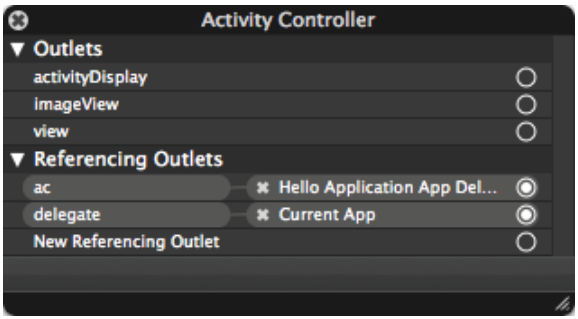
14.6 Der File's Owner

Ein Vorteil kleiner, konzentrierter *.nib*-Dateien wie `ActivityView.xib` besteht darin, dass Sie den *File's Owner* auswählen können, der am besten passt. Hier ist die *.nib*-Datei momentan der View, den wir im Hauptfenster ausgeben wollen, sodass unser *File's Owner* dafür verantwortlich sein wird, den View und seine Inhalte zu verwalten. Die natürliche Wahl für den *File's Owner* ist das `ActivityController`-Objekt, das wir in `MainMenu.xib` erzeugt haben.

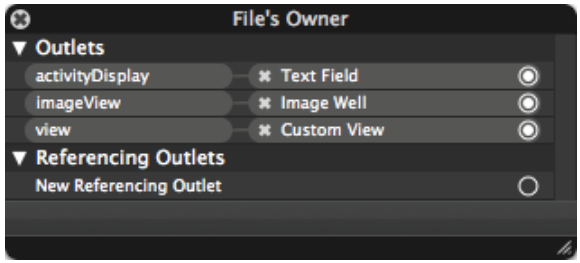
Um den *File's Owner* festzulegen, wählen Sie *File's Owner* im Dokumentenfenster in `ActivityView.xib`. Verwenden Sie den Identity Inspector, um den Klassennamen für den *File's Owner* mit `ActivityController` festzulegen, und speichern Sie ab.



Lassen Sie uns den `ActivityController` verknüpfen. Es gibt fünf mögliche Verbindungen. Die beiden Referenz-Outlets sind in `MainMenu.xib` verbunden.



Wir müssen die drei Outlets in `ActivityView.xib` verknüpfen. Verbinden Sie das `view`-Outlet mit dem `Custom View`, das `imageView`-Outlet mit dem `NSImageView` und das `activityDisplay`-Outlet mit dem `NSTextView`.



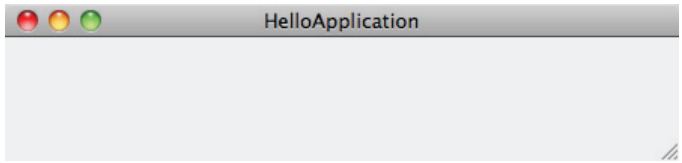
Ihr `ActivityController`-Objekt hat ein Bein in beiden Nibs, Sie können also Verbindungen im ersten und im zweiten Nib herstellen. Das `ActivityController`-Objekt ist die Verbindung zwischen diesen beiden Welten.

Ich möchte hier einen wichtigen Punkt hervorheben: Der `ActivityController` des *File's Owner* ist ein Proxy für die `ActivityController`-Instanz, die Sie im `MainMenu`-Nib erzeugt haben. Es handelt sich um dasselbe Objekt.

Hätten Sie stattdessen einen `ActivityController` in das `ActivityView`-Nib gezogen, wäre eine neue Instanz des `ActivityController` erzeugt worden. Das wären zwei separate Objekte geworden, die Verbindungen im einen und im anderen Nib hätten also der Kommunikation nicht weitergeholfen.

14.7 Übung: Den View laden

Klicken Sie *Build & Run* an. Der Code sollte kompiliert, gelinkt und ausgeführt werden, und ein leeres Fenster sollte erscheinen.



Das wollen wir ändern. Welches Objekt kann mit dem Fenster und dem View-Controller kommunizieren? Nutzen Sie dieses Objekt, um den Content-View des Fensters als View des View-Controllers festzulegen. Wir könnten das im IB erledigen, aber wir wollen es mit Code lösen.

Sie wissen, dass Sie die richtige Lösung gefunden haben, wenn Sie *Build & Run* anklicken und das Textfeld und das Icon im Fenster erscheinen.

14.8 Lösung: Den View laden

Der Anwendungs-Delegate ist das Objekt mit den Eigenschaften für das Fenster und den View-Controller. Sie müssen also nur die `ActivityController`-Header-Datei importieren und den Content-View mit dem View des View-Controllers festlegen:

MultipleNibs/HelloApplication19/HelloApplicationAppDelegate.m

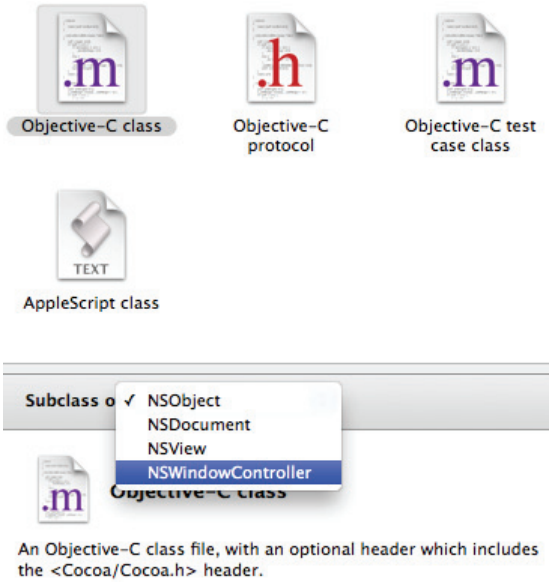
```
#import "HelloApplicationAppDelegate.h"
► #import "ActivityController.h"

@implementation HelloApplicationAppDelegate

@synthesize window, ac, currentApp;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
►     self.window.contentView = ac.view;
}
@end
```

Klicken Sie *Build & Run* an, und die Anwendung sollte so laufen wie gegen Ende des vorigen Kapitels.[1]

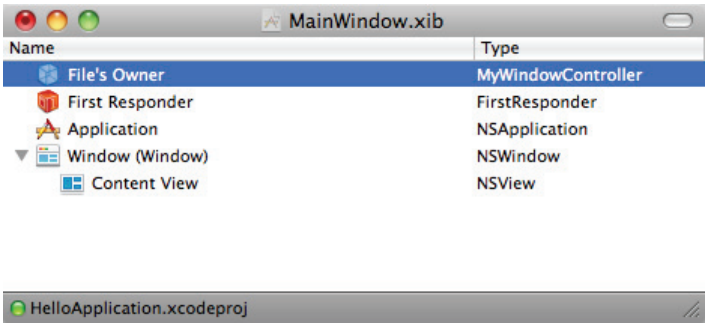
14.9 Das Window-Nib anlegen

Wir wollen das `MainMenu`-Nib noch einmal aufteilen. Dazu erzeugen wir ein neues Nib, das das Fenster enthält. Lassen Sie uns einen Window-Controller anlegen, den wir als *File's Owner* verwenden können. In Xcode wählen Sie den Ordner `Classes` in *Groups & Files* und legen eine neue Datei vom Typ `Mac OS X > Cocoa Class > Objective-C Class` an. Nutzen Sie die Drop-down-Liste, um den `NSWindowController` auszuwählen. Nennen Sie die Klasse `MyWindowController`.

1 Zum Anlegen und Nutzen des neuen Nib sind viele Schritte notwendig. Sie können sich aber immer den herunterladbaren Code ansehen, um diese Version mit der vorherigen zu vergleichen.



Nun wählen Sie den Ordner `Resources` und legen eine neue Datei mit der Vorlage `Mac OS X > User Interface > Window XIB` an. Nennen Sie das Nib `MainWindow.xib`. Verwenden Sie den Identity Inspector, um den Typ des *File's Owner* mit `MyWindowController` festzulegen.



Verwenden Sie den Attributes Inspector, um für den Titel des Fensters *Hello Application* festzulegen. Verbinden Sie das `window`-Outlet des File's Owner mit Ihrem Fenster und speichern Sie ab. Wir kommen gleich auf dieses Nib zurück, wollen aber zuerst unser `MainMenu`-Nib sichern und aufräumen.

14.10 Das Window-Nib laden

Öffnen Sie `MainMenu.xib`. Löschen Sie das `CurrentApp-`, das `Activity-Controller-` und das `NSWindow`-Objekt zusammen mit dem `Content-View`. Viel ist nicht mehr übrig – nur der App-Delegate, das Menü und ein paar Standardelemente.

Der App-Delegate besitzt keine Verbindung mehr zu dem Modell, dem Fenster oder dem View-Controller. Sie werden gleich sehen, wie man diese Objekt wieder verbindet. Jetzt wollen wir erst mal die Header-Datei für den App-Delegate aufräumen:

MultipleNibs/HelloApplication20/HelloApplicationAppDelegate.h

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

@interface HelloApplicationAppDelegate : NSObject <NSApplicationDelegate> {
}

@end
```

In der Implementierungsdatei instanziieren wir den Window-Controller und laden seine *.nib*-Datei:

MultipleNibs/HelloApplication20/HelloApplicationAppDelegate.m

```
#import "HelloApplicationAppDelegate.h"
► #import "MyWindowController.h"

@implementation HelloApplicationAppDelegate

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
►     [[MyWindowController alloc] initWithWindowNibName:@"MainWindow"];
}
@end
```

`initWithWindowNibName:` erwartet den Namen des Nib als `NSString`. Es gibt während der Kompilierung also keine Prüfung, ob der Name korrekt geschrieben wurde. Sie geben den Namen des Nib, nicht aber seine Erweiterung (`.nib` oder `.xib`) an. Wenn Sie *Build & Run* anklicken, wird

die Anwendung kompiliert, gelinkt und gestartet, doch das Anwendungsfenster erscheint nicht. Das wollen wir als Nächstes korrigieren.

14.11 Das Fenster präsentieren

Überschreiben Sie die `initWithWindowNibName:`-Methode in `MyWindowController.m`. Klicken Sie auf *Build & Run*, und das Anwendungsfenster erscheint, sobald die Anwendung gestartet wird. Es gibt aber noch einiges zu tun. Wir müssen den View und das Modell so miteinander verknüpfen, dass wir die startenden und endenden Anwendungen sehen können.

MultipleNibs/HelloApplication20/MyWindowController.m

```
#import "MyWindowController.h"

@implementation MyWindowController

-(id) initWithWindowNibName:(NSString *)windowNibName {
    if (self =[super initWithWindowNibName:windowNibName]) {
►       [self showWindow:self];
    }
    return self;
}

@end
```

14.12 Übung: View und Modell verknüpfen

Sie kennen bereits eine Möglichkeit, um den View mit dem Modell zu verknüpfen: Sie können ein `ActivityController`- und ein `CurrentApp`-Objekt in das `MainWindow.xib` einfügen, entsprechende Outlets in der Header-Datei des Window-Controllers eintragen und den Content-View des Fensters in der Implementierung des Window-Controllers festlegen.

Wir wollen einen anderen Ansatz verfolgen. Es gibt keinen Grund, den View-Controller und das Modell in unsere *.nib*-Datei einzufügen, weshalb wir das programmtechnisch lösen wollen. Das ähnelt stark dem, was wir getan haben, als wir programmtechnisch den Window-Controller angelegt und das Nib festgelegt haben.

Fügen Sie Eigenschaften für den View-Controller und das Modell in `MyWindowController.h` ein. In der Implementierungsdatei erzeugen Sie das `ActivityController`-Objekt und initialisieren es mit `initWithNibName:bundle:`. Der Nib-Name lautet `ActivityView`, und Sie müssen `nil` für das Bundle übergeben. Erzeugen Sie auch eine Instanz von `CurrentApp` und legen Sie den `ActivityController` als ihren Delegate fest. Legen Sie den Content-View des Fensters mit dem View des `Acti-`

vityController fest. Wenn Sie wollen, können Sie herauszufinden versuchen, wie man die Größe des Fensters so anpassen kann, dass es den View genau umschließt.

Klicken Sie auf *Build & Run*. Keine Panik! Es sieht so aus, als würde nichts passieren. Das Fenster sollte erscheinen und Sie sollten das Textfeld und die Bildquelle sehen, doch Sie sehen nichts. `HelloApplication` hat den Start abgeschlossen, bevor das Modell sich für Notifikationen registrieren konnte. Starten Sie iCal. Das Icon und die Meldung erscheinen wie erwartet. Versuchen Sie, das Problem zu beheben, damit das Icon und die Meldung für `HelloApplication` beim Programmstart erscheinen.

14.13 Lösung: View und Modell verknüpfen

Wir haben Eigenschaften in die Header-Datei eingefügt:

MultipleNibs/HelloApplication21/MyWindowController.h

```
#import <Cocoa/Cocoa.h>
@class CurrentApp;
@class ActivityController;

@interface MyWindowController : NSWindowController {
    CurrentApp *currentApp;
    ActivityController *ac;
}
@property CurrentApp *currentApp;
@property ActivityController *ac;

@end
```

Sehen wir uns die Implementierung genauer an:

MultipleNibs/HelloApplication21/MyWindowController.m

```
#import "MyWindowController.h"
#import "ActivityController.h"
#import "CurrentApp.h"

@implementation MyWindowController

@synthesize ac, currentApp;

-(void) setUpView {
    self.ac = [[ActivityController alloc]
                initWithNibName:@"ActivityView" bundle:nil];
    self.currentApp = [[CurrentApp alloc] init];
    self.currentApp.delegate = self.ac;
    [self.window setContentSize:[self.ac.view bounds].size];
    self.window.contentView = self.ac.view;
    [self.ac applicationDidLaunch:
                        [NSRunningApplication currentApplication]];
```

```
17  }
18
19  -(id) initWithWindowNibName:(NSString *)windowNibName {
20      if (self =[super initWithWindowNibName:windowNibName]) {
21          [self setUpView];
22          [self showWindow:self];
23      }
24      return self;
25  }
26  @end
```

- Wir haben den `ActivityController` instanziiert und das zu kontrollierende Nib geladen (Zeilen 10 und 11).
- Wir instanziieren das `CurrentApp`-Objekt in der nächsten Zeile und legen den `ActivityController` als seinen Delegate fest.
- In Zeile 14 passen wir die Größe des Fensters so an, dass sein Content-Bereich genau der Größe des Views entspricht.
- Sobald wir die richtige Größe haben, ersetzen wir den Content-View durch den View des View-Controllers.
- Schließlich initialisieren wir in Zeile 16 den View-Controller, damit er das Icon und den App-Namen von *HelloApplication* benutzt.

Wir besitzen nun drei separate Nibs, die eine zusammenhängende Einheit bilden und einfach zu verstehen sind. Das erste, `MainMenu.xib`, besteht aus der App, dem App-Delegate und dem (ungenutzten) Menü. Das zweite Nib wird geladen, und sein *File's Owner* wird durch den App-Delegate erzeugt. Das zweite Nib, `MainWindow.xib`, enthält das Fenster und verwendet den Window-Controller als *File's Owner*. Der Window-Controller lädt das dritte Nib und erzeugt dessen *File's Owner* – den `ActivityController` zusammen mit dem Modell. Das dritte Nib, `ActivityView.xib`, enthält den View und verwendet den View-Controller als dessen *File's Owner*.

In diesem Kapitel wurden viele Schritte vorgestellt. Die Technik ist wichtig. Seien Sie bei *.nib*-Dateien genauso wachsam wie bei Ihren Klassen. Wenn sie zu groß werden und Sie den Überblick verlieren, sollten Sie über eine Aufteilung nachdenken. Es gibt zwei Hauptgründe für die Aufteilung einer *.nib*-Datei. Zum einen soll sie zusammenhängend einer einzigen Aufgabe dienen. Zum anderen können Sie (wie Sie in Kapitel 17, *Daten auf Festplatte speichern*, auf Seite 273 noch sehen werden) den Speicher besser verwalten, indem sie das Nib nur bei Bedarf laden und den Speicher wieder freigeben, sobald Sie es nicht mehr benötigen.

Kapitel 15

Eigene Views entwickeln

Bisher haben wir den Interface Builder benutzt, um unseren gesamten View zu entwickeln. In diesem Kapitel unternehmen wir den ersten Schritt in Richtung einer komplizierteren Benutzerschnittstelle. Sie werden die im Interface Builder festgehaltenen Informationen um Programmcode erweitern und eine Subklasse von `NSView` anlegen, um Ihren eigenen View zu entwickeln.

Wir beginnen damit, das Icon des startenden bzw. endenden Programms in einem `NsimageView` zu zeichnen. Dann werden wir einen eigenen View aufbauen und und ansehen, wie man Grafiken und Text darin zeichnet. Schließlich verschieben wir noch die Icon-Ausgabe in unseren neuen View, damit Sie sehen, wie man Images darin ausgibt.

15.1 Einen eigenen View anlegen

Legen Sie eine neue *.nib*-basierte Datei für unsere Zeichenoperationen an und nennen Sie sie `IconView.xib`. Wir benötigen einen View-Controller für dieses Nib, weshalb wir auch ein neues `NSObject` namens `IconViewController` anlegen. Ändern Sie die Header-Datei so ab, dass sie `NSViewController` erweitert und das `ActivityMonitorDelegate`-Protokoll implementiert.

CustomView/HelloApplication22/IconViewController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface IconViewController :NSViewController <ActivityMonitorDelegate>{
}
@end
```

Das `ActivityMonitorDelegate`-Protokoll verlangt zwei Methoden, die wir in `IconViewController.m`: schon einmal anlegen wollen:

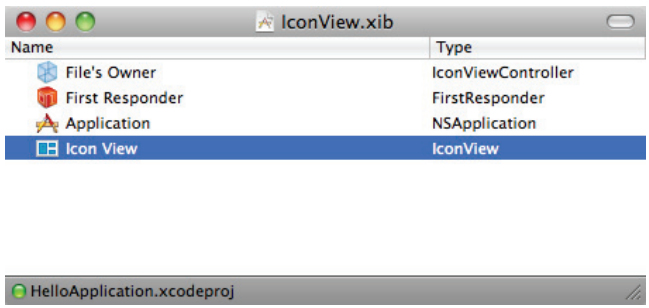
CustomView/HelloApplication22/IconViewController.m

```
#import "IconViewController.h"

@implementation IconViewController

-(void) applicationDidLaunch: (NSRunningApplication *) app {}
-(void) applicationDidTerminate: (NSRunningApplication *) app {}

@end
```

Nun kommt der neue Teil.

Wir bauen eine eigene Klasse für den View auf. Legen Sie eine neue Klasse von `NSView` an und nennen Sie sie `IconView`.[1] Wir werden in diesem Kapitel meist an der `IconView`-Klasse arbeiten, aber zuerst wollen wir die zugehörige Infrastruktur aufbauen.

In `IconView.xib` verwenden Sie den Identity Inspector, um die Klasse des *File's Owner* mit `IconViewController` und die Klasse des Views mit `IconView` festzulegen. Verwenden Sie den Connections Inspector, um das View-Outlet des `IconViewController` mit dem `IconView` zu verbinden. Benutzen Sie außerdem den Size Inspector, um die Breite des `IconView` auf 216 und seine Höhe auf 240 Pixel festzulegen. Speichern Sie ab und beenden Sie den IB.



1 Wählen Sie das Objective-C-Klassentemplate und verwenden Sie die Option `NSView` aus der Drop-down-Liste.

Wir müssen auch den `MyWindowController` anpassen, damit ein `IconViewController` erzeugt und das `IconView`-Nib geladen wird. In der `MyWindowController`-Header-Datei können wir einfach alle Instanzen von `ActivityController` in `IconView-Controller` ändern.

CustomView/HelloApplication22/MyWindowController.h

```
#import <Cocoa/Cocoa.h>
@class CurrentApp;
@class IconViewController;

@interface MyWindowController : NSWindowController {
    CurrentApp *currentApp;
    IconViewController *ac;
}
@property CurrentApp *currentApp;
@property IconViewController *ac;

@end
```

Die einzigen beiden Änderungen an der Implementierung bestehen im Import von `IconViewController.h` und Ändern der Zeile, in der der View-Controller erzeugt und das Nib geladen wird:

CustomView/HelloApplication22/MyWindowController.m

```
self.ac = [[IconViewController alloc]
           initWithNibName:@"IconView" bundle:nil];
```

Klicken Sie auf *Build & Run*, und ein leeres Fenster präsentiert stolz unseren neuen View. In einer Minute werden wir dieses Fenster mit etwas füllen.

15.2 Formen in einem eigenen View zeichnen

Bevor wir Grafiken oder Text in unseren View zeichnen, müssen wir uns darüber im Klaren sein, wie `NSView` sich selbst zeichnet. Das unterscheidet sich deutlich von dem, was Sie bisher gesehen haben.

Als wir im vorigen Kapitel ein Icon in einen Image-View gezeichnet haben, wurde die Nachricht `setImage:` an eine Instanz von `NSImageView` gesendet. Und wenn ein Text in einem Textfeld erscheinen sollte, haben Sie die Nachricht `setStringValue:` an eine Instanz von `NSTextField` gesendet. Das ist wahrscheinlich ein Schema, das Sie von anderen Plattformen her kennen. Sie rufen die verschiedenen Konfigurations- und Zeichenmethoden für das zu zeichnende Objekt auf.

In einen `NSView` zu zeichnen, ist völlig anders. Sie legen alle Parameter und Instruktionen für die Zeichenoperationen innerhalb der `NSView`-Methode `drawRect:` fest. Sie rufen `drawRect:` nicht direkt auf. Die

Methode wird aufgerufen, wenn der View erstmals gezeichnet werden muss. Gleich zeige ich Ihnen noch, wie Sie ihm signalisieren, dass er den View neu zeichnen muss.

Zunächst wollen wir nur ein großes blaues Rechteck innerhalb des Views erzeugen:

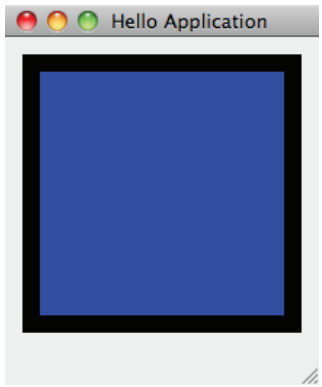
CustomView/HelloApplication23/IconView.m

```
#import "IconView.h"

@implementation IconView

-(void)drawRect:(NSRect)dirtyRect {
►    NSRect innerRect = NSMakeRect(18, 42, 180, 180);
►    [[NSColor blueColor] set];
►    [NSBezierPath fillRect:innerRect];
}

@end
```

Auf den ersten Blick sieht das seltsam aus. Sie senden die `set`-Nachricht an das `NSColor`-Objekt. *Eigentlich* sollte man doch den `NSView` anweisen, die Farbe festzulegen, oder? Bei Views müssen Sie daran denken, dass sie sich selbst zeichnen. Wenn sie neu gezeichnet werden müssen, übernimmt der zu zeichnende View den Fokus, seine `drawRect:`-Methode wird aufgerufen und der Fokus wird wieder freigegeben.

Das Zeichnen innerhalb der `drawRect:`-Methode können Sie sich so vorstellen: Wenn Sie `set` an `[NSColor blueColor]` senden, sagen Sie so etwas wie: „Ich habe dich auserwählt, blauer Buntstift, komm in meine Hand!“ Wenn Sie dann in der nächsten Zeile das Rechteck ausfüllen, liegt der blaue Buntstift in Ihrer Hand und Sie benutzen ihn.

Stellen Sie sich vor, Sie wollen einen schwarzen Rand um das blaue Rechteck zeichnen. Wenn wir für das Rechteck die gleiche Größe verwenden und eine Pinseldicke von 12 Punkten wählen, sieht unser Rechteck so aus:



Beachten Sie, dass unter dem Rechteck mehr Platz ist als darüber. Das liegt daran, dass sich das Koordinatensystem an der unteren linken Ecke unseres Views orientiert. Die x-Achse steigt nach rechts hin an und die y-Achse von unten nach oben. Wir haben also ein Rechteck konstruiert, das am Punkt (18, 42) beginnt und 180 Punkte breit und hoch ist. Der „Pinselstrich" setzt aber (wenn man nichts anderes festlegt) mittig an, die untere linke Ecke des schwarzen Rahmens liegt also am Punkt (12, 36).

Ein Großteil der Arbeit mit grafischen Objekten erinnert eher an C als an Objective-C. So ist dieses `NSRect` beispielsweise ein Struct, das ein Rechteck repräsentiert. Es besteht aus zwei anderen Structs: einem `NSPoint`, der die Position des Rechtecks festlegt, sowie einer `NSSize`, die seine Breite und Höhe angibt.

Beachten Sie auch, dass `NSMakeRect()` eine normale C-Funktion ist, keine Objective-C-Methode. Sie werden solche Konstruktorfunktionen für die meisten verwendeten Structs vorfinden.

Sobald Sie das Rechteck erzeugt haben, können Sie es sowohl für die blaue Fläche als auch für den schwarzen Rahmen verwenden. Nachdem Sie das blaue Rechteck gefüllt haben, sagen Sie `[[NSColor blackColor] set]` und zeichnen mit Ihrem schwarzen Stift etwas anderes. Statt `llRect:` benutzen wir diesmal `strokeRect:`, um einen schwarzen Rahmen um unser blaues Rechteck zu zeichnen.

CustomView/HelloApplication24/IconView.m

```
-(void)drawRect:(NSRect)dirtyRect {
    NSRect innerRect = NSMakeRect(18, 42, 180, 180);
    [[NSColor blueColor] set];
    [NSBezierPath fillRect:innerRect];
►   [[NSColor blackColor] set];
►   [NSBezierPath setDefaultLineWidth:12];
►   [NSBezierPath strokeRect:innerRect];
}
```

Sehen Sie sich die Dokumentation zu `NSBezierPath` an, und Sie werden sehen, dass Sie damit alle Arten von Formen zeichnen können. Wir werden bei Rechtecken bleiben, wenn wir unseren View für *HelloApplication* entwickeln.

Bisher zeichnen wir das Rechteck nur, wenn der View zum ersten Mal instanziiert wird. Als Nächstes wollen wir den View jedesmal neu zeichnen, wenn eine Anwendung startet oder endet.

15.3 Übung: Die Pinselfarbe ändern

Sie sollen einen roten Rahmen zeichnen, wenn eine Anwendung endet, und beim Start einen grünen. Verschwenden Sie keine Zeit darauf, das Rechteck zu füllen.

Fügen Sie eine Eigenschaft namens `alertColor` vom Typ `NSColor` in Ihren `IconView` ein. Ändern Sie den `drawRect:`-Code so ab, dass die Pinselfarbe in `alertColor` geändert und das Rechteck umrahmt wird.

Implementieren Sie `applicationDidLaunch:` im `IconViewController` so, dass er den `alertColor` von `IconView` auf grün setzt. Die `IconView`-Instanz ist der View des `IconViewController`. Unglücklicherweise müssen Sie ein Typecasting der Variablen `view` zum Typ `IconView` vornehmen, um die `alertColor`-Eigenschaft setzen zu können.

Ein weiterer Schritt ist nötig: Sie müssen angeben, dass der `IconView` verändert wurde, damit er neu gezeichnet wird. Sie signalisieren dem `view`, das er neu gezeichnet werden muss, indem Sie ihm folgende Nachricht senden:

```
[self.view setNeedsDisplay:YES]
```

Implementieren Sie die `applicationDidTerminate:`-Methode und setzen Sie *alertColor* auf rot. Klicken Sie auf *Build & Run*. Wenn die Anwendung gestartet ist, sehen Sie einen grünen Rahmen. Wenn Sie eine andere Anwendung beenden, wird der Rahmen rot.

15.4 Lösung: Die Pinselfarbe ändern

Wenn Sie die Eigenschaft `alertColor` hinzufügen, müssen Sie ihr Speicherattribut mit `copy` angeben, weil `NSColor` dem `NSCopying`-Protokoll entspricht. Ich habe auch ein `NSRect` namens `frameRect` als Instanzvariable eingefügt.

CustomView/HelloApplication25/IconView.h

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

@interface IconView : NSView {
    NSColor *alertColor;
    NSRect frameRect;
}
@property(copy) NSColor *alertColor;
@end
```

In der Implementierungsdatei des Views senden wir die `set`-Methode an `alertColor` und ziehen einen Rahmen um das Rechteck. Ich habe das Rechteck und die Pinselbreite in `awakeFromNib` angelegt, damit ich das nicht bei jedem Aufruf von `drawRect:` machen muss.

CustomView/HelloApplication25/IconView.m

```
#import "IconView.h"

@implementation IconView

@synthesize alertColor;

-(void)drawRect:(NSRect)dirtyRect {
    [self.alertColor set];
    [NSBezierPath strokeRect:frameRect];
}

-(void) awakeFromNib {
    frameRect = NSMakeRect(18, 42, 180, 180);
    [NSBezierPath setDefaultLineWidth:12];
}
@end
```

Im `IconViewController` legen wir die Farbe des Frames an und teilen dem View mit, dass er sich selbst neu zeichnen muss.

CustomView/HelloApplication25/IconViewController.m

```
#import "IconViewController.h"
#import "IconView.h"

@implementation IconViewController

-(void) applicationDidLaunch: (NSRunningApplication *) app {
    ((IconView *)self.view).alertColor = [NSColor greenColor];
    [self.view setNeedsDisplay:YES];
}
-(void) applicationDidTerminate: (NSRunningApplication *) app {
    ((IconView *)self.view).alertColor = [NSColor redColor];
    [self.view setNeedsDisplay:YES];
}
@end
```

Natürlich kann diese Art des Zeichnens sehr komplex werden. Sie können sehr komplizierte Pfade aus einer kleinen Menge von Primitiven aufbauen. Sie können sehr viele Parameter (wie Pinselbreite und Farbe) anpassen. Wenn Sie so weit sind, sollten Sie sich ein gutes Buch zu Quartz besorgen. Zwei gute sind *Programming with Quartz, 2D and PDF Graphics in Mac OS X* [GL06] und *Quartz 2D graphics for Mac OS X developers* [Tho06].

Ein Grund dafür, dass MVC so schwer zu verstehen ist, ist, dass der Controller und das Modell oder der Controller und der View häufig vermischt werden. Sie werden einige Aspekte eines Controllers sogar in Dingen sehen, die wir eigentlich als View-Komponenten betrachten, etwa Buttons. Hier können Sie eine klare Trennung der Verantwortlichkeiten zwischen dem `IconViewController` und dem `IconView` feststellen.

15.5 Grafiken zeichnen

Wie wollen das Icon der Anwendung in einem Subview unseres Views ausgeben, damit es innerhalb unseres Rahmens erscheint. Sie werden aber bemerken, dass das überraschend einfach ist. Wir erzeugen einen Image-View, spezifizieren seine Größe und legen ihn als Subview unseres Views fest.

CustomView/HelloApplication26/IconView.m

```
-(void) awakeFromNib {
    frameRect = NSMakeRect(18, 42, 180, 180);
    [NSBezierPath setDefaultLineWidth:12];
►   imageRect = NSMakeRect(36, 66, 144, 144);
►   self.imageView = [[NSImageView alloc] initWithFrame:imageRect];
►   [self addSubview:self.imageView];
}
```

Wie Sie sehen, benötigen wir eine Eigenschaft namens `imageView` und eine Instanzvariable namens `imageRect`.

CustomView/HelloApplication26/IconView.h

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

@interface IconView : NSView {
    NSColor *alertColor;
    NSRect frameRect;
►   NSRect imageRect;
►   NSImageView *imageView;
}
@property(copy) NSColor *alertColor;
► @property NSImageView *imageView;
@end
```

Wie zuvor habe ich den sich wiederholenden Code in zwei Delegate-Methoden refaktoriert. Ich habe außerdem die hervorgehobene Zeile eingefügt, um das Icon der aktiven Anwendung als Inhalt unserer `imageView`-Eigenschaft `image` festzulegen.

CustomView/HelloApplication26/IconViewController.m

```
#import "IconViewController.h"
#import "IconView.h"

@implementation IconViewController

-(void) displayColor:(NSColor *) color for:(NSRunningApplication *) app {

►       ((IconView *)self.view).alertColor = color;
        ((IconView *)self.view).imageView.image = app.icon;
        [self.view setNeedsDisplay:YES];
}
-(void) applicationDidLaunch: (NSRunningApplication *) app {
        [self displayColor:[NSColor greenColor] for:app];
}
-(void) applicationDidTerminate: (NSRunningApplication *) app {
        [self displayColor:[NSColor redColor] for:app];
}
@end
```

Leider ist die eigentliche Grafik zu klein. Wenn `drawRect:` aufgerufen wird, gibt es die Grafik in ihrer normalen Größe aus. Ich möchte aber, dass sie den Großteil des Rahmens ausfüllt. Darum habe ich noch eine Zeile eingefügt, die die Größe des Icons in der `drawRect:`-Methode anpasst.

CustomView/HelloApplication26/IconView.m

```
-(void)drawRect:(NSRect)dirtyRect {
        [self.alertColor set];
        [NSBezierPath strokeRect:frameRect];
►       [self.imageView.image setSize:imageRect.size];
}
```

Klicken Sie auf *Build & Run*. Während Sie Anwendungen starten und beenden, sollten Sie in etwa das hier sehen:

Lehnen Sie sich ein wenig zurück und denken Sie darüber nach, was Sie getan haben, um eine Grafik zu zeichnen. Sie haben einen Container für die Grafik instanziiert und ihre Lage und Größe festgelegt. Sie haben den Container mit der Grafik gefüllt. Ja, die Sache ist tatsächlich so einfach.

15.6 Text zeichnen

Wenn Sie viel Text ausgeben, sollten Sie ein Textfeld oder einen TextView verwenden. Das ist bei uns nicht der Fall. Wir geben eine einfache, kurze Meldung aus und können den Text daher in unserem View zeichnen. Dieser Abschnitt zeigt Ihnen die Technik, die Sie verwenden sollten, um Text in einem eigenen View zu zeichnen.

Um einen String in einem eigenen View zu zeichnen, senden wir die Nachricht `drawinRect:withAttributes:` innerhalb der `drawRect:`-Methode an den String. Die Methode verlangt zwei Parameter. Der erste enthält die Lage und Größe des Rechtecks, in dem der String gezeichnet werden soll. Der zweite ist ein Dictionary mit allen zu setzenden Attributen.

Fügen Sie eine Instanzvariable für das Rechteck und für den String (der den Namen der Anwendung enthalten wird) in `IconView.h` ein. Legen Sie eine Eigenschaft für die `appName`-Variable an:

CustomView/HelloApplication27/IconView.h

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

@interface IconView : NSView {
    NSColor *alertColor;
    NSRect frameRect;
    NSImageView *imageView;
    NSRect imageRect;
►   NSString *appName;
►   NSRect textRect;
}
@property(copy) NSColor *alertColor;
@property NSImageView *imageView;
► @property(copy) NSString *appName;
@end
```

In `IconView.m` fügen Sie die folgende Zeile am Ende der `awakeFromNib`-Methode hinzu, um das Rechteck zu erzeugen:

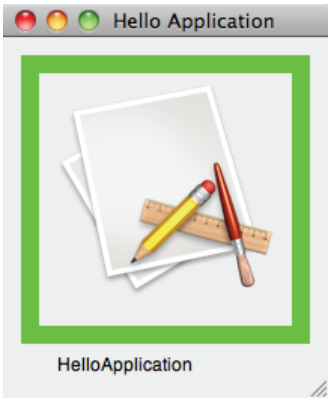
CustomView/HelloApplication27/IconView.m

```
textRect = NSMakeRect(36, 10, 144, 20);
```

Auf diese Weise erzeugen Sie das Rechteck nur einmal und schreiben bei jedem Aufruf der `drawRect:`-Methode einen anderen Text hinein. Bei dieser Version verwenden wir das Standardaussehen, weshalb wir `nil` anstelle eines Dictionary übergeben. Fügen Sie die folgende Zeile am Ende der `drawRect:`-Methode hinzu:

```
[self.appName drawInRect:textRect withAttributes:nil];
```

Sie sollten nun den „Standard“-Look sehen:



Sie können das Aussehen dieses Strings verändern, indem Sie Attribute in ein mutables Dictionary schreiben. Zum Beispiel könnten wir das Dictionary mit den folgenden Attributen füllen, um den String blau, fett und zentriert auszugeben:[2]

CustomView/HelloApplication28/IconView.m

```
NSMutableParagraphStyle *par = [[NSMutableParagraphStyle alloc] init];
[par setAlignment:NSCenterTextAlignment];
textAttributes = [[NSMutableDictionary alloc] initWithObjectsAndKeys:
                  [NSColor blueColor], NSForegroundColorAttributeName,
                  par, NSParagraphStyleAttributeName,
                  [NSFont boldSystemFontOfSize:12], NSFontAttributeName,
                  nil];
```

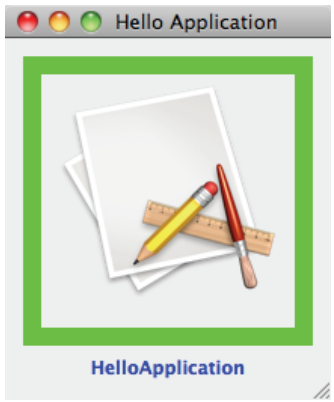
2 Eine Liste der Standardattribute für die Textausgabe finden Sie unter: *http://developer.apple. com/documentation/Cocoa/Reference/ApplicationKit/Classes/NSAttributedString_ AppKitAdditions/ Reference/Reference.html#//apple_ref/doc/uid/20000167-BAJJCCFC*. Sie können in der Dokumentation aber auch einfach nach NSAttributedString(AppKitAdditions) suchen.

Es ist bemerkenswert wenig Code notwendig, um diesen Effekt in unserem View und seinem Controller zu implementieren. Hier die abschließende Implementierung des View-Controllers:

CustomView/HelloApplication28/IconViewController.m

```
#import "IconViewController.h"
#import "IconView.h"

@implementation IconViewController

-(void) displayColor:(NSColor *) color for:(NSRunningApplication *) app {
     ((IconView *)self.view).alertColor = color;
     ((IconView *)self.view).imageView.image = app.icon;
     ((IconView *)self.view).appName = app.localizedName;
     [self.view setNeedsDisplay:YES];
}

-(void) applicationDidLaunch: (NSRunningApplication *) app {
     [self displayColor:[NSColor greenColor] for:app];
}
-(void) applicationDidTerminate: (NSRunningApplication *) app {
    [self displayColor:[NSColor redColor] for:app];
}
@end
```

Das Ergebnis sollte so aussehen:



Die Header-Datei für unseren `IconView` ist gewachsen. Wir haben viele Instanzvariablen eingefügt, aber nur die drei, die vom View-Controller gesetzt werden müssen, wurden als Eigenschaften preisgegeben.

CustomView/HelloApplication28/IconView.h

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

@interface IconView : NSView {
    NSColor *alertColor;
    NSRect frameRect;
```

```
    NSImageView *imageView;
    NSRect imageRect;
    NSString *appName;
    NSRect textRect;
    NSMutableDictionary *textAttributes;
}
@property(copy) NSColor *alertColor;
@property NSImageView *imageView;
@property(copy) NSString *appName;
@end
```

Ich habe `IconView.m` refaktoriert, um die Initialisierung der drei verschiedenen Teile zu trennen. Es ist uns auch gelungen, die `drawRect:`-Methode klein und einfach zu halten.

CustomView/HelloApplication28/IconView.m

```
#import "IconView.h"

@implementation IconView

@synthesize alertColor, imageView, appName;

-(void)drawRect:(NSRect)dirtyRect {
     [self.alertColor set];
     [NSBezierPath strokeRect:frameRect];
     [self.imageView.image setSize:imageRect.size];
     [self.appName drawInRect:textRect withAttributes:textAttributes];
}
-(void) setUpFrameRect {
     frameRect = NSMakeRect(18, 42, 180, 180);
     [NSBezierPath setDefaultLineWidth:12];
}
-(void) setUpImageView {
    imageRect = NSMakeRect(36, 66, 144, 144);
    self.imageView = [[NSImageView alloc] initWithFrame:imageRect];
    [self addSubview:self.imageView];
}
-(void) setUpTextView {
    textRect = NSMakeRect(36, 10, 144, 20);
    NSMutableParagraphStyle *par = [[NSMutableParagraphStyle alloc] init];
    [par setAlignment:NSCenterTextAlignment];
    textAttributes = [[NSMutableDictionary alloc] initWithObjectsAndKeys:
                      [NSColor blueColor], NSForegroundColorAttributeName,
                      par, NSParagraphStyleAttributeName,
                      [NSFont boldSystemFontOfSize:12], NSFontAttributeName,
                      nil];
}
-(void) awakeFromNib {
     [self setUpFrameRect];
     [self setUpImageView];
     [self setUpTextView];
}
@end
```

Sie haben nun also einen eigenen View entwickelt, in dem Sie Formen, Grafiken und Text zeichnen können. Sie haben gesehen, wie man visuelle Objekte im Programmcode erzeugt und konfiguriert. In diesem Kapitel haben wir uns auf den View konzentriert. Im nächsten Kapitel wollen wir eine Tabelle aufbauen, die eine Liste aller laufenden Anwendungen enthält.

Kapitel 16

Daten in einer Tabelle darstellen

Von jetzt an werden wir uns bis zum Ende dieses Buches durch immer coolere und leistungsfähigere Möglichkeiten zur Interaktion mit Daten arbeiten. In diesem Kapitel werden Sie eine ziemlich schlichte Möglichkeit kennenlernen, Daten in einer Tabellenansicht darzustellen.

Wir werden ein Array als Datenquelle für unseren Tabellen-View verwenden. Jeder Eintrag in diesem Array besteht aus einer `NSRunningApplication`-Instanz, mit deren Hilfe wir eine Zeile der von uns ausgegebenen Tabelle füllen. Die Objekteigenschaften werden den Tabellenspalten entsprechen. Diese Übereinstimmung zwischen Bezeichnern und Eigenschaften ist der erste Schritt auf Ihrem Weg durch *Key Value Coding*, *Key Value Observing* und die restlichen Bindungen hin zu *Core Data*.

Gegen Ende dieses Kapitels reagiert Ihr Tabellen-View auf Updates der zugrunde liegenden Daten, und Sie werden in der Lage sein, die Daten im Tabellen-View zu aktualisieren. Wir beginnen mit der Entwicklung eines Tabellen-Views und verknüpfen diesen mit einer simulierten Datenquelle.

16.1 Tabellen und Datenquellen

Lassen Sie uns mit der Einrichtung eines Tabellen-Views und der dazugehörigen Datenquelle anfangen. Im ersten Beispiel füllen Sie jede Zelle der Tabelle mit der dazugehörigen Zeilen- und Spaltennummer auf. Wir können für dieses Beispiel das `ActivityView`-Nib und den `ActivityController` wiederverwenden.

Zuerst räumen Sie `ActivityController.m` auf:

Tables/HelloApplication29/ActivityController.m

```
#import "ActivityController.h"

@implementation ActivityController

-(void) applicationDidLaunch: (NSRunningApplication *) app {}
-(void) applicationDidTerminate: (NSRunningApplication *) app {}
@end
```

Entfernen Sie auch die Eigenschaften und Instanzvariablen aus der Header-Datei:

Tables/HelloApplication29/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface ActivityController : NSViewController <ActivityMonitorDelegate> {
}
@end
```

Wir müssen den von `MyWindowController` geladenen View-Controller und das Nib ändern. Wir wollen daher die Header-Datei korrigieren, um uns dieses Hin und Her ein wenig zu vereinfachen. Der Typ der Variablen `ac` wird so geändert, dass er weder ein `ActivityController` noch ein `IconViewController` ist. Stattdessen deklarieren wir ihn als `NSViewController` und legen fest, dass er das `ActivityMonitorDelegate`-Protokoll implementiert.

Tables/HelloApplication29/MyWindowController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"
@class CurrentApp;

@interface MyWindowController : NSWindowController {
    CurrentApp *currentApp;
    NSViewController <ActivityMonitorDelegate> *ac;
}
@property CurrentApp *currentApp;
@property NSViewController <ActivityMonitorDelegate> *ac;

@end
```

In der Implementierungsdatei ändern wir die Instanziierung von `ac` entsprechend und fügen die passende `import`-Anweisung ein:

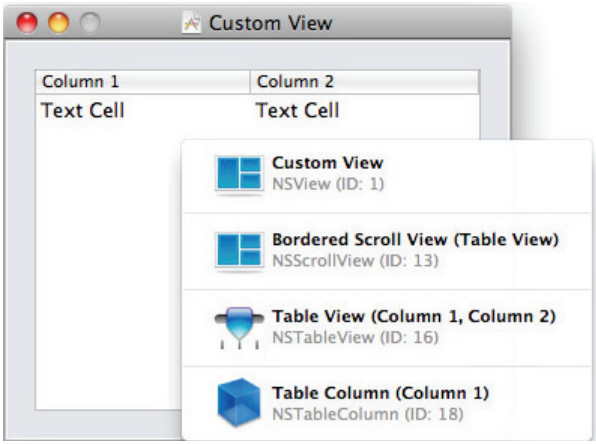
Tables/HelloApplication29/MyWindowController.m

```
self.ac = [[ActivityController alloc]
           initWithNibName:@"ActivityView" bundle:nil];
```

Klicken Sie `ActivityView.xib` doppelt an. Wenn der Interface Builder die *.nib*-Datei öffnet, löschen Sie das Textfeld und die Bildquelle aus dem View. Passen Sie die Größe des Views so an, dass zwei Spalten und ungefähr ein halbes Dutzend Zeilen Platz finden. Ziehen Sie einen Tabellen-View (table view) aus der Library in den View und positionieren Sie ihn mithilfe der Richtlinien. Tatsächlich ist diese Komponente ein `NSTable-View` innerhalb eines `NSScrollView`. Die Tabelle ist mit zwei Spalten vorkonfiguriert. Sie können diesen Wert mithilfe des Attributes Inspectors nach oben oder unten korrigieren. Klicken Sie die Kopfzeile jeder Spalte doppelt an und nennen Sie die linke Spalte *Column 1* und die rechte *Column 2*.

Wählen sie den Tabellen-View und verbinden Sie sowohl das Datenquellen- (`dataSource`) als auch das `delegate`-Outlet mit dem `Activity-Controller`.[1]

Leider ist das Anklicken des Tabellen-Views gar nicht so einfach. Man glaubt, es ausgewählt zu haben, und findet sich stattdessen im Scroll-View wieder.[2] Also versuchen Sie es mit einem Doppelklick und müssen erkennen, dass Sie eine Spalte ausgewählt haben – nicht so ganz das, was Sie wollten. Navigieren Sie sich stattdessen im Dokumentenfenster durch die Hierarchie, oder halten Sie `Shift`- und `Control`-Taste gedrückt und klicken Sie die Tabelle an. Sie sehen so etwas wie das hier:



1 Der `ActivityController` ist der *File's Owner*.
2 Den Scroll-View brauchen Sie so gut wie nie.

Nun können Sie den Tabellen-View schnell auswählen und die Verbindungen herstellen. Speichern Sie ab. Sie haben einen Teil des Grundgerüsts für den Tabellen-View und seine Datenquelle aufgebaut. Doch es klaffen noch zwei Lücken, die es zu füllen gilt. Lassen Sie uns die in einer kleinen Übung untersuchen.

16.2 Übung: Eine einfache Datenquelle implementieren

Klicken Sie *Build & Run* an. Sie sehen die folgende Meldung:

```
sIllegal NSTableView data source (<ActivityController: 0x200053b60>).
Must implement numberOfRowsInTableView: and
              tableView:objectValueForTableColumn:row:
```

Das ist so hilfreich, wie eine Fehlermeldung nur sein kann: Sie teilt uns mit, welche Methoden in `ActivityController` implementiert werden müssen, damit er als Datenquelle für unseren Tabellen-View fungieren kann. Ich möchte das zuerst Ihnen als Übungsaufgabe überlassen.

Die Zeilen einer Tabelle repräsentieren verschiedene Datensätze Ihrer Datenquelle und die Spalten die verschiedenen Attribute der einzelnen Datensätze. Zum Beispiel könnten Sie eine Liste der Mitglieder Ihrer lokalen CocoaHeads-Gruppe ausgeben. Die erste Spalte könnte den Vornamen und die zweite den Nachnamen enthalten. Die Datenquelle könnte natürlich noch weitere Informationen enthalten (E-Mail-Adressen, Telefonnummern usw.), die Sie in der Tabelle darstellen wollen.

Sie haben die Spalten im Interface Builder festgelegt, als Sie den Tabellen-View erzeugten, aber üblicherweise kennen Sie die Anzahl der Datensätze bis zur Laufzeit nicht. Wenn Sie eine Anwendung entwickeln, bei der Datensätze hinzugefügt und gelöscht werden können, ändert sich die Zahl sogar während der Laufzeit. Daher muss die Datenquelle der Tabelle mitteilen können, wie viele Zeilen der Tabellen-View enthält. Der Tabellen-View kann sich dann selbst auffüllen, indem er die Anzahl sichtbarer Zeilen und Spalten durchläuft und die Datenquelle für jede Zelle fragt, was denn da hineingehört. Das geschieht mithilfe der Methode `tableView:objectValueForTableColumn:row:`.

Die Datensätze der Datenquelle werden häufig in einem Array gespeichert, aber eigentlich kann sie in jeder beliebigen Form vorliegen, solange die Datenquelle auf die beiden Methoden `numberOfRowsInTableView:` und `tableView:objectValueForTableColumn:row:` reagiert. Das läuft darauf hinaus, dass eine Indexierung der Datensätze über Integerwerte möglich sein muss.

Implementieren Sie diese Methoden in der einfachstmöglichen Art und Weise. Geben Sie in `numberOfRowsInTableView:`.einen festen Integerwert zurück. Wählen Sie eine ausreichend große Zahl wie 100, damit auch der Scroll-View einbezogen wird. Geben Sie bei `tableView:objectValueForTableColumn:row:` die Zeilennummer zurück. Beide Spalten der Zeile geben die Zeilennumer aus.

Sehen Sie in der Dokumentation nach, damit Sie die richtigen Methodensignaturen verwenden. Suchen Sie in der Dokumentation nach den Namen der einzelnen Methoden. Sobald Sie eine Methode gefunden haben, kopieren Sie die Methodensignatur aus der Dokumentation und fügen sie in Ihren Code ein. Was immer Sie auch tun, tippen Sie es nicht selbst ein!

Oha! Wir haben noch ein anderes Problem: Es sollte nicht allzu schwer sein, die Zeilennummer zurückzuliefern, aber wie finden Sie die Spaltennummer? Sehen Sie sich die Dokumentation zu `NSTableColumn` an, und Sie finden Methoden wie `identier`, `dataCellForRow:` und `headerCell`. Es sieht nicht so aus, als würde die Spalte ihre eigene Indexnummer kennen. Sie werden gleich sehen, warum. Für's Erste füllen Sie jede Zeile mit der Zeilennummer aus.

16.3 Lösung: Eine einfache Datenquelle implementieren

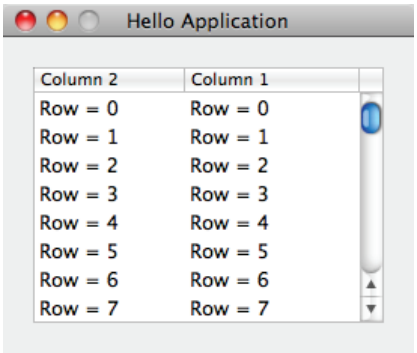
Sie sollten keinerlei Änderungen an der `ActivityController`-Header-Datei vorgenommen haben. Wenn man es sich mal kurz genau überlegt, mag das überraschend sein. Sie müssen nicht deklarieren, dass der `ActivityController` die Protokolle `NSTableViewDataSource` oder `NSTableViewDelegate` implementiert. Sie können sie hinzufügen, wenn Sie wollen – wahrscheinlich wird anderen Lesern Ihres Quelltexts dadurch der Zweck des `ActivityController` deutlicher. Sie müssen es aber nicht, weil nur der Tabellen-View die in diesem Protokoll deklarierten Methoden aufruft, weshalb Sie auch keine Warnungen erhalten.

In der Implementierungsdatei sollten Sie diese Methoden eingefügt haben:

Tables/HelloApplication30/ActivityController.m

```
-(NSInteger)numberOfRowsInTableView:(NSTableView *)aTableView {
    return 100;
}
-(id)           tableView:(NSTableView *)aTableView
objectValueForTableColumn:(NSTableColumn *)aTableColumn
                      row:(NSInteger)rowIndex {
    return [NSString stringWithFormat:@"Row = %d", rowIndex];
}
```

Klicken Sie *Build & Run* an. In der laufenden Anwendung klicken Sie die erste Spalte doppelt an, um sie auszuwählen. Ziehen Sie die Spalte nach rechts, bis die beiden Spalten die Plätze tauschen, wie in der nachfolgenden Abbildung zu sehen ist.



Wir können und dürfen Spalten also nicht über ihre Positionen bestimmen, weil sich die Position ändern kann. Üblicherweise hätten wir in der Überschrift auch keine Nummern verwendet, aber ich wollte Ihnen deutlich machen, dass die beiden die Positionen getauscht haben.

16.4 Übung: Eine Datenquelle einführen

Im Moment besitzen wir noch keine realen Daten für unsere Tabelle. Lassen Sie uns ein Array der Applikationen erzeugen, die beim Start unserer Anwendung laufen, und es in der Tabelle ausgeben. Für den Augenblick wollen wir in beiden Spalten die gleichen Informationen ausgeben.

Deklarieren Sie ein veränderliches Array namens `runningApps` als Eigenschaft des `Activity-Controller`. Überschreiben Sie `initWithNibName:Bundle:` so, dass der Wert dieses Arrays auf die `runningApplications` der gemeinsam genutzten Instanz von `NSWorkspace` gesetzt wird.[3]

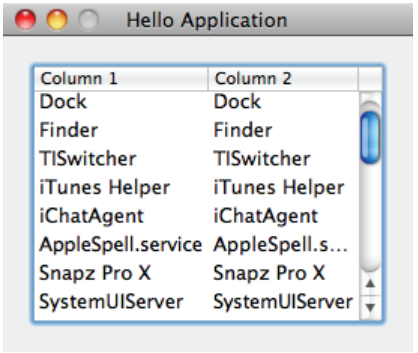
Ihr Tabellen-View soll den Inhalt des `runningApps`-Arrays ausgeben. Wie viele Zeilen hat die Tabelle? Die Antwort auf diese Frage liefert der Wert, den Sie in `numberOfRowsInTableView` zurückgeben.

Sie müssen nur noch jede Zelle mit dem Namen der Anwendung der einzelnen Datensätze auffüllen. Das `runningApps`-Array besteht aus Objekten vom Typ `NSRunningApplication`. Sie wissen, wie man den

3 Es ist in der Dokumentation als Klassenmethode gekennzeichnet, aber als diese Zeilen geschrieben wurden, war es als Instanzmethode implementiert.

Namen der Anwendung ermittelt. Sie müssen aus dem `runningApps`-Array das Objekt abfragen, das denselben Index hat wie `rowIndex`.

Ihre Anwendung sollte dann so aussehen:



16.5 Lösung: Eine Datenquelle einführen

Fügen Sie die Eigenschaft `runningApps` in Ihre Header-Datei ein. Es hört sich seltsam an, aber wir müssen als Speicherattribut `retain` angeben. Üblicherweise bekommen wir das kostenlos, indem wir gar kein Attribut angeben: Damit würden wir das Standardattribut `assign` erhalten, das unter der Garbage Collection mit `retain` identisch ist. Das Problem besteht hier darin, dass `NSMutableArray` `NSArray` erweitert, das wiederum mit `copy` arbeitet. Daher müssen wir `retain` explizit angeben, damit der Compiler sich sicher sein kann, dass wir wissen, was wir tun:

Tables/HelloApplication31/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface ActivityController : NSViewController <ActivityMonitorDelegate,
                                NSTableViewDelegate, NSTableViewDataSource> {
    NSMutableArray *runningApps;
}
@property(retain) NSMutableArray *runningApps;
@end
```

Als Nächstes springen wir rüber zur Implementierungsdatei. Wie viele Zeilen soll unsere Tabelle besitzen? Sie sollte so viele Spalten besitzen, wie es laufende Anwendungen gibt. Wenn Ihre Tabelle über ein Array gefüllt wird, sollte die Zahl der Zeilen im Tabellen-View immer der aktuellen Anzahl von Zeilen im Array entsprechen.

Tables/HelloApplication31/ActivityController.m

```
-(NSInteger)numberOfRowsInTableView:(NSTableView *)aTableView {
    return [runningApps count];
}
```

Was soll für die Zellen in einer bestimmten Zeile zurückgegeben werden? Der `localizedName` des `RunningApplication`-Objekts mit dem gleichen Index.

Tables/HelloApplication31/ActivityController.m

```
-(id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
                          row:(NSInteger)rowIndex {
    return [[runningApps objectAtIndex:rowIndex] localizedName];
}
```

Der schwierige Teil dieser Übung ist die Initialisierung von `runningApps`. Wir können das `runningApplications`-Array nicht einfach in ein veränderliches Array umwandeln. Stattdessen erzeugen wir ein veränderliches Array und fügen dann den Inhalt von `runningApplications` ein.

Tables/HelloApplication31/ActivityController.m

```
-(id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)nibBundle{
    if (self =[super initWithNibName:nibName bundle:nibBundle] ) {
        self.runningApps = [NSMutableArray arrayWithCapacity:20];
       [self.runningApps addObjectsFromArray:[[NSWorkspace sharedWorkspace]
                                              runningApplications]];
    }
    return self;
}
```

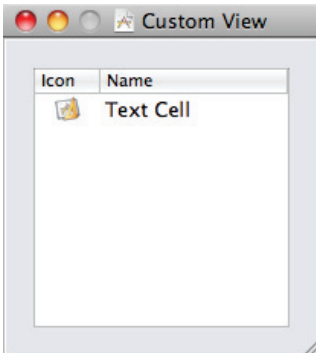
Nun wollen wir uns zur nächsten Stufe übergehen und in jeder Spalte etwas anderes ausgeben.

16.6 Zellen basierend auf Spaltenüberschriften füllen

Der Rumpf der `tableView:objectValueForTableColumn:row:`-Methode kann sehr schnell sehr hässlich werden. Wie kann man beispielsweise das Icon der Anwendung in der ersten und ihren Namen in der zweiten Spalte ausgeben?

Zuerst müssen wir die *.nib*-Datei so ändern, dass Grafiken in der ersten Spalte angezeigt werden können. Öffnen Sie `ActivityView.xib` im Interface Builder. Suchen Sie in der Library nach `NSImageCell`. Ziehen Sie es über die erste Spalte. Das Wort *TextCell* sollte durch das generische Anwendungs-Icon ersetzt werden.

Ändern Sie den Titel der ersten Spalte in *Icon* und den zweiten in *Name*:

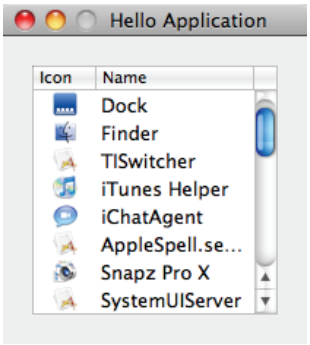


Sie könnten versucht sein, eine `if/else`-Konstruktion wie diese hier zu verwenden:

Tables/HelloApplication32/ActivityController.m

```
//Kommen Sie nicht auf die Idee, es so zu machen!
- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
                          row:(NSInteger)rowIndex {
►   if([[[aTableColumn headerCell] title] isEqualToString:@"Icon"]){
►       return [[self.runningApps objectAtIndex:rowIndex] icon];
►   } else {
►   return [[self.runningApps objectAtIndex:rowIndex] localizedName];
►   }
}
```

Die gute Nachricht ist, dass der Code genau das macht, was wir wollen. Wir sehen die Icons der Anwendung auf der linken und ihre Namen auf der rechten Seite.

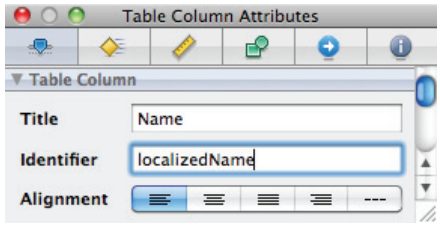


Die schlechte Nachricht ist, dass der Code unglaublich hässlich ist. Wir wollen nicht jedesmal eine `if`-Anweisung verwenden, wenn wir eine Zelle in einer Tabelle auffüllen. Wir haben in einem anderen Teil unseres Codes ein Dictionary benutzt, um ein `if` zu eliminieren. Diesmal wollen wir eine andere Technik verwenden.

16.7 Spaltenbezeichner als Schlüssel

Statt die Spaltenüberschrift in unserer `if`-Anweisung zu verwenden, können wir eine andere Eigenschaft von Tabellenspalten nutzen: die sogenannten *Identier* (Bezeichner). Der Titel ist der Text, der in der Überschriftenzelle einer Spalte sichtbar ist. Der Text soll für den Benutzer der Anwendung verständlich sein. Der Bezeichner ist für den Benutzer unsichtbar. Die Datenquelle verwendet ihn, um die Spalte zu identifizieren. Wir können den Identifier als Schlüssel in einem Dictionary oder als Name einer Eigenschaft oder Methode verwenden.

Wählen Sie also die erste Spalte im Interface Builder aus und verwenden Sie den Attributes Inspector, um *icon* als `Identier` anzugeben. In gleicher Weise setzen Sie den `Identier` in der zweiten Spalte auf *localizedName*. Ich habe diese Namen nicht zufällig gewählt. Das sind die Namen der Eigenschaften in der `NSRunning-Application`-Klasse, die in den beiden Spalten erscheinen sollen.



Nun können Sie die Methode mit dem gleichen Namen wie der `identier` aufrufen. Das ist einfach der Getter für die Eigenschaft mit diesem Namen.

Tables/HelloApplication33/ActivityController.m

```
- (id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
                          row:(NSInteger)rowIndex {
►   return [[self.runningApps objectAtIndex:rowIndex]
►       performSelector:NSSelectorFromString([aTableColumn identifier])];
}
```

Was muss also passieren, damit die Tabelle mit Icons und Namen gefüllt wird? Zuerst fragt der Tabellen-View seine Datenquelle, aus wie vielen Zeilen diese Tabelle besteht. Die Datenquelle antwortet mit `[runningApps count]`. Dann fragt der Tabellen-View für jede Zelle in der Tabelle: „Was gehört hier rein?“

Die Datenquelle antwortet nun mit der Information für die Anwendung, deren Index im Array der Zeilennummer entspricht.[4] Und in jeder Zelle antwortet die Datenquelle mit dem Wert der Eigenschaft, die dem Spalten-Identifier der Zelle entspricht. Ganz schön elegant.

Sehen Sie sich an, wie das den Code vereinfacht: Es gibt keine Entscheidungen zu treffen. Besser noch, wir müssen den Code nicht ändern, wenn wir neue Spalten einführen.

16.8 Ausblick auf bevorstehende Knüller

Es hat sich herausgestellt, dass ein Vorteil von Eigenschaften darin besteht, dass sie bestimmten Namenskonventionen folgen, die es uns erlauben, Objekte zu behandeln, als wären sie Dictionaries. Mit anderen Worten können wir den Namen einer Eigenschaft, deren Wert wir abrufen wollen, genau so verwenden, wie wir Schlüssel bei einem Dictionary benutzen. Sie werden darüber in Kapitel 19, *Key Value Coding*, auf Seite 303 sehr viel mehr erfahren.

Das ist jetzt so eine hervorragende Stelle, um diese Technik anzuwenden, dass ich nicht abwarten kann. Sie müssen nicht wissen, wie Key Value Coding (kurz: KVC) funktioniert, um diesen Anwendungsfall würdigen zu können. Statt den `identier` in einen `SEL` umzuwandeln und diesen Selektor dann auszuführen, benutzen wir die `valueForKey:`-Methode und übergeben den `identier` als Schlüssel.

Tables/HelloApplication34/ActivityController.m

```
-(id)tableView:(NSTableView *)aTableView
    objectValueForTableColumn:(NSTableColumn *)aTableColumn
                          row:(NSInteger)rowIndex {
►   return [[self.runningApps objectAtIndex:rowIndex]
►                       valueForKey:[aTableColumn identifier]];
}
```

Das ist noch so etwas an Cocoa, das mir ein Lächeln auf die Lippen zaubert.

16.9 Übung: Zeilen einfügen und löschen

Unser Minidock zeigt nun einen Schnappschuss der Icons und Namen der Anwendungen, die gerade laufen, wenn unsere Anwendung gestartet wird. Nun wollen wir der Liste startende Anwendungen hinzufügen und endende Anwendungen aus ihr löschen.

4 Glücklicherweise beginnt die Zählung jeweils bei 0.

Wir könnten die Liste jedesmal über das runningApplications-Array von NSWorkspace runningApplications neu generieren, wenn eine Anwendung startet oder endet. Ich möchte Ihnen hier aber zeigen, wie Sie Daten in die zugrunde liegende Datenquelle eines Tabellen-Views einfügen bzw. sie löschen und den View dann aktualisieren können.

Sie besitzen bereits Delegate-Methoden, die reagieren, sobald eine Anwendung gestartet oder beendet wird. Implementieren Sie eine, die eine startende Anwendung an das Ende des Tabellen-Views anhängt, indem Sie einen Eintrag an das Ende von runningApps anhängen und den Tabellen-View aktualisieren. Implementieren Sie die andere so, dass sie eine beendete Anwendung aus dem Tabellen-View entfernt, indem sie den entsprechenden Eintrag aus runningApps entfernt und den Tabellen-View aktualisiert.

16.10 Lösung: Zeilen einfügen und löschen

Wir müssen den Tabellen-View aus unserem ActivityController heraus kontaktieren. Fügen Sie ein Outlet in die Header-Datei ein:

Tables/HelloApplication35/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface ActivityController : NSViewController <ActivityMonitorDelegate,
NSTableViewDelegate, NSTableViewDataSource> {
    NSMutableArray *runningApps;
►   NSTableView *table;
}
@property(retain) NSMutableArray *runningApps;
► @property IBOutlet NSTableView *table;
@end
```

Im Interface Builder verbinden Sie dieses Outlet mit dem Tabellen-View.

In Xcode sollten Sie zwei Methoden in die Implementierungsdatei für den ActivityController eingefügt haben. Dann fügen Sie den entsprechenden Eintrag in das runningApps-Array ein (bzw. löschen ihn) und weisen den Tabellen-View an, sich neu zu zeichnen.[5]

Tables/HelloApplication35/ActivityController.m

```
-(void) applicationDidLaunch: (NSRunningApplication *) app {
     [self.runningApps addObject:app];
     [self.table reloadData];
}
```

5 Denken Sie auch daran, dass beim Einfügen einer Eigenschaft in eine Header-Datei eine Synthetisierung in der entsprechenden Implementierungsdatei notwendig ist.

```
-(void) applicationDidTerminate: (NSRunningApplication *) app {
    [self.runningApps removeObject:app];
    [self.table reloadData];
}
```

16.11 Zeilen manuell entfernen

Bisher haben wir Änderungen an der Datenquelle initiiert, die sich in der Tabelle widerspiegelten. In diesem Abschnitt wollen wir uns die Tatsache zunutze machen, dass der `ActivityController` auch der Delegate für den Tabellen-View ist.

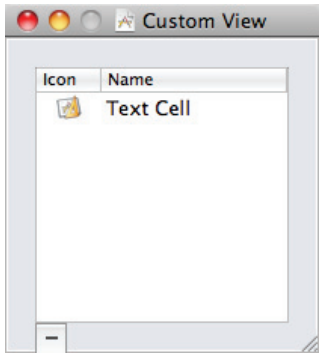
Lassen Sie uns einen Remove-Button in unsere Anwendung aufnehmen. Er soll es dem Benutzer erlauben, Anwendungen aus der Liste zu entfernen. Der Button wird nur sichtbar, wenn der Benutzer eine Anwendung im Tabellen-View auswählt. Klickt der Benutzer den Button an, wird die entsprechende Anwendung aus der Liste entfernt und der Button verschwindet wieder.

Zuerst fügen wir ein Outlet und eine Aktion in die Header-Datei des `ActivityController` ein. Das Outlet wird mit dem neuen Button verbunden; die Aktion wird die Methode sein, die aufgerufen wird, wenn man den Button anklickt.

Tables/HelloApplication36/ActivityController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"

@interface ActivityController : NSViewController <ActivityMonitorDelegate,
NSTableViewDelegate, NSTableViewDataSource> {
    NSMutableArray *runningApps;
    NSTableView *table;
►   NSButton *deleteButton;
}
@property(retain) NSMutableArray *runningApps;
@property IBOutlet NSTableView *table;
► @property IBOutlet NSButton *deleteButton;
► -(IBAction)removeRow:(id)sender;
@end
```

Im Interface Builder fügen Sie einen kleinen quadratischen Button unter dem Tabellen-View ein. Statt ihm einen Titel zu geben, lassen wir ihn ein Minuszeichen ausgeben. Dazu öffnen Sie den Attributes Inspector und legen als Grafik für den Button `NSRemove-Template` fest. Aktivieren Sie außerdem die Checkbox *Hidden*.



Verbinden Sie das `deleteButton`-Outlet des `ActivityController` mit dem Button und verknüpfen Sie ihn gleichzeitig mit der Aktion `removeRow:`. Speichern Sie ab.

Sie müssen nun zwei Methoden in `ActivityController` implementieren. Zuerst eine Delegate-Methode, die aufgerufen wird, wenn der Benutzer eine Zeile auswählt. Wir wollen nur erreichen, dass der Button sichtbar wird, wenn eine Zeile ausgewählt wird.

Tables/HelloApplication36/ActivityController.m

```
-(void)tableViewSelectionDidChange:(NSNotification *)notification {
     [self.deleteButton setHidden:NO];
}
```

Wenn der Benutzer den Button anklickt, entfernen wir die Zeile aus dem `runningApps`-Array und laden den Tabellen-View neu. Wir stellen außerdem sicher, dass im Tabellen-View keine Zeile mehr gewählt ist und der Button wieder versteckt wird.

Tables/HelloApplication36/ActivityController.m

```
-(IBAction)removeRow:(id)sender{
     [self.runningApps removeObjectAtIndex:[self.table selectedRow]];
     [self.table deselectAll:nil];
     [self.table reloadData];
     [self.deleteButton setHidden:YES];
}
```

Auch hier finde ich es wunderbar, dass sich der Code wie eine Beschreibung dessen liest, was wir zu erreichen versuchen.

Klicken Sie *Build & Run* an, und Sie sind in der Lage, Zeilen auszuwählen und aus der Liste zu entfernen. Wenn Sie `HelloApplication` beenden und wieder starten, sind die gelöschten Einträge aber wieder da. Im nächsten Kapitel erweitern wir unser Beispiel um Persistenz und Präferenzen.

Kapitel 17

Daten auf Festplatte speichern

Am Ende des vorigen Kapitels haben wir es den Benutzern ermöglicht, Anwendungen aus dem Tabellen-View zu entfernen. Wenn Sie im Büro ein Spielchen spielen oder im Netz surfen, möchten Sie nicht unbedingt, dass das Spiel oder der Webbrowser in der Liste laufender Anwendungen auftaucht.

Doch so wie es im Moment programmiert ist, tauchen die gelöschten Elemente wieder auf, wenn Sie *HelloApplication* erneut ausführen. In diesem Kapitel werden Sie die Liste der entfernten Elemente auf zwei Arten festhalten. Sie werden auch lernen, wie man mit Präferenzen arbeitet, damit der Benutzer entscheiden kann, ob er die Liste verwenden will.

Wie immer ist das Beispiel nur Mittel zum Zweck. Wir sehen uns das Erzeugen, Sichern und Lesen von Eigenschaftslisten an. Sie werden erfahren, wie man Objektgraphen archiviert, speichert und dearchiviert. Am Ende des Kapitels werden Sie lernen, wie man zwei Arten von Standardeinstellungen erzeugt und einliest.

Beginnen wollen wir damit, die Liste entfernter Anwendungen in unserer laufenden Anwendung zu speichern.

17.1 Während der laufenden Anwendung speichern

Bevor wir uns darum Gedanken machen, wie wir uns in *HelloApplication* die gelöschten Elemente alter Programmläufe merken können, müssen wir, während die Anwendung läuft, wissen, was die entfernten Elemente waren. Starten Sie *HelloApplication*, und ich zeige Ihnen, was ich meine.

Entfernen Sie eine Anwendung aus der Liste – ich werde iCal als Beispiel benutzen. iCal läuft gerade, aber es steht nicht auf der Liste. Also beenden Sie iCal und starten es erneut. Sehen Sie das Problem? Es erscheint wieder in der Liste der laufenden Anwendungen. Ich möchte iCal nicht bei jedem Start des Programms wieder aus der Liste entfernen müssen.

Um dieses Problem zu lösen, wollen wir eine Helferklasse entwickeln, um nachzuhalten, welche Anwendungen wir entfernt haben. Erzeugen Sie eine neue Klasse namens `BanishedApps`. Verwenden Sie dazu die `NSObject`-Vorlage. Wir werden die Liste der entfernten Anwendungen in einem veränderlichen `NSMutableArray` vorhalten. Wir benötigen außerdem Methoden zum Hinzufügen von Anwendungen zu diesem Array und zur Prüfung, ob eine Anwendung bereits enthalten ist.

Persistence/HelloApplication37/BanishedApps.h

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

@interface BanishedApps : NSObject {
    NSMutableArray *apps;
}
@property(retain) NSMutableArray *apps;
-(void)add:(NSRunningApplication *) app;
-(BOOL)contains:(NSRunningApplication *) app;
@end
```

Zur Implementierung gibt es nicht viel zu sagen. Wir initialisieren das veränderliche Array in der `init`-Methode. Die Methoden `add:` und `contains:` sind einfache Wrapper, die es uns erlauben, Objekte vom Typ `NSRunningApplication` zu übergeben und die Information mithilfe des `localizedName` der Anwendung als String zu speichern.

Persistence/HelloApplication37/BanishedApps.m

```
#import "BanishedApps.h"

@implementation BanishedApps
```

```
@synthesize apps;

-(BOOL)contains:(NSRunningApplication *) app {
    return [self.apps containsObject:app.localizedName];
}
-(void)add:(NSRunningApplication *) app {
    if ([self contains:app]) return;
    [self.apps addObject:app.localizedName];
}
-(id)init {
    if (self =[super init]) {
        self.apps = [NSMutableArray arrayWithCapacity:5];
    }
    return self;
}
@end
```

Die erste Zeile der `add:`-Methode könnte ein wenig seltsam anmuten.

Möglicherweise sind Sie eher daran gewöhnt, die Methode so zu formulieren:

```
-(void) add: (NSRunningApplication *) app {
    if (![self contains:app]) {
        [self.apps addObject:app.localizedName];
    }
}
```

Diese Version liest sich wie folgt: „Wenn diese Anwendung nicht in unserer Liste entfernter Anwendungen steht, dann füge sie in die Liste ein.“ Mein Ansatz ist stattdessen, die Methode möglichst schnell wieder zu verlassen. Wenn die Anwendung bereits in unserer Liste steht, gibt es für uns nichts mehr zu tun.

Wir müssen einige Änderungen an `ActivityController.m` vornehmen. Fügen Sie die folgende Zeile in `initWithNibName:bundle:` ein, um das `BanishedApps`-Objekt zu initialisieren.[1]

Persistence/HelloApplication37/ActivityController.m

```
self.banishedApps = [[BanishedApps alloc] init];
```

Wenn die Anwendung startet, müssen Sie die folgende Prüfung durchführen, bevor Sie eine Anwendung zu `runningApps:` hinzufügen:

1 Sie wissen bereits, dass diese Zeile eine Reihe weiterer Änderungen am Code nach sich zieht. Sie müssen die Instanzvariable `banishedApps` sowie die dazugehörige Eigenschaft angelegt haben. Sie müssen die Eigenschaft synthetisiert haben. Abschließend müssen Sie eine Vorwärtsdeklaration für die `BanishedApps`-Klasse in der `ActivityController`-Header-Datei angegeben und eine entsprechende Import-Anweisung am Anfang der Implementierungsdatei eingefügt haben.

Persistence/HelloApplication37/ActivityController.m

```
-(void) applicationDidLaunch: (NSRunningApplication *) app {
►    if ([self.banishedApps contains:app]) return;
    [self.runningApps addObject:app];
    [self.table reloadData];
}
```

Wenn der Benutzer eine Anwendung aus der Tabelle entfernt, müssen wir die folgende Zeile einfügen, um sie in die Liste gelöschter Anwendungen aufzunehmen:

Persistence/HelloApplication37/ActivityController.m

```
-(void)removeRow:(id)sender{
►    [self.banishedApps add:[self.runningApps
►                            objectAtIndex:[self.table selectedRow]]];
    [self.runningApps removeObjectAtIndex:[self.table selectedRow]];
    [self.table deselectAll:nil];
    [self.table reloadData];
    [self.deleteButton setHidden:YES];
}
```

Klicken Sie auf *Build & Run*. Entfernen Sie eine Anwendung. Beenden Sie die Anwendung und starten Sie sie neu. Diesmal erscheint sie nicht wieder in der Tabelle. Cool.

Wir haben nicht allzu viele Änderungen vorgenommen, aber jetzt funktioniert alles, solange *HelloApplication* läuft. Nun wollen wir die Sache um Persistenz erweitern, damit wir nicht bei jedem Start von *HelloApplication* dieselben Anwendungen entfernen müssen.

17.2 Wo man Support-Dateien ablegt

Wir wollen die Namen gelöschter Anwendungen jedesmal auf der Festplatte speichern, wenn eine Anwendung hinzugefügt wird. Und wir wollen den Inhalt dieser Datei einlesen, wenn `BanishedApps` instanziiert wird. In diesem Abschnitt werden wir mit einer Eigenschaftsliste (property list, Endung *.plist*) arbeiten, die wir aus dem `NSMutableArray` erzeugen. Diese Technik eignet sich hervorragend für kleine, aus Strings und Zahlen bestehende Datenmengen.

Sie speichern die Dateien, die Ihre Anwendung unterstützen, die sogenannten *Support-Dateien*, in einem Ordner, der nach der Anwendung oder dem Unternehmen benannt ist und in `~/Library/Application Support` liegt. Das `~` ist ein Kürzel für Ihr Home-Verzeichnis. Wir nutzen die Funktion `NSSearchPathFor-DirectoriesInDomains()`, um den Pfad für das Support-Verzeichnis der Anwendung zu bestimmen:

```
NSSearchPathForDirectoriesInDomains(NSApplicationSupportDirectory,
                                                    NSUserDomainMask, YES)
```

Der erste Parameter ist eine Konstante, die angibt, dass wir das Support-Verzeichnis für die Anwendung suchen. Im zweiten Parameter übergeben wir ebenfalls eine Konstante. Diese gibt an, dass wir das `Library`-Verzeichnis des Benutzers verwenden wollen, nicht das des Systems. Der dritte Parameter ist ein auf `YES` gesetzter boolescher Wert, der festlegt, dass ~ in den vollständigen Pfad für das Benutzerverzeichnis aufgelöst werden soll. Dieser Aufruf gibt ein Array zurück, in dem der Pfad als einziges Element enthalten ist. Wir ziehen diesen Wert als String aus dem Array heraus und hängen `/HelloApplication` an, um den vollständigen Pfad zum Speichern der Anwendungsdaten zu erzeugen.

Wir erledigen diese Arbeit in `BanishedApps.m`. Der Code sieht nicht gerade schön aus, aber es handelt sich um Standardcode zur Lokalisierung der Support-Dateien einer Anwendung.

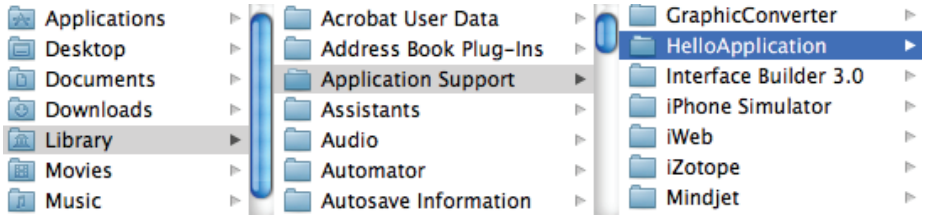
Persistence/HelloApplication38/BanishedApps.m

```
-(void) setSupportFile {
    NSFileManager *fileManager = [NSFileManager defaultManager];
    NSString *appSupport =
        [NSSearchPathForDirectoriesInDomains(
                                    NSApplicationSupportDirectory,
                                    NSUserDomainMask, YES)
        objectAtIndex:0];
    NSString *dir =
      [NSString stringWithFormat:@"%@/HelloApplication", appSupport];
    [fileManager createDirectoryAtPath:dir
           withIntermediateDirectories:YES
                            attributes:nil
                                 error: nil];
    self.dataFile =
        [dir stringByAppendingPathComponent:@"removedApps.plist"];
}
```

Fügen Sie die Eigenschaft `NSString *dataFile` in die `BanishedApps`-Header-Datei ein. Ihnen wird aufgefallen sein, dass Sie für `NSString`-Eigenschaften immer das Attribut `copy` angeben müssen. Wenn Sie das vergessen, werden Sie über eine entsprechende Compilerwarnung darauf aufmerksam gemacht.

Sie müssen außerdem einen Aufruf von `[self setSupportFile];` in die `init`-Methode einfügen.

Klicken Sie *Build & Run* an. Öffnen Sie `~/Library/Application Support/`. Dort sollten Sie ein neu erzeugtes Verzeichnis namens `HelloApplication` sehen.



17.3 Speichern in einer Plist

Objekte vom Typ `NSArray` wissen, wie sie sich selbst in eine *.plist*-Datei schreiben können. Tatsächlich kann sich jede Objektkollektion vom Typ `NSDictionary`, `NSArray`, `NSString`, `NSDate`, `NSData` und `NSNumber` ganz einfach selbst als Plist auf der Festplatte sichern. Andere Objekte lassen sich entsprechend fit machen, indem man sie in `NSData` umwandelt und in ein Array oder Dictionary packt.

Wir wollen den Inhalt des `apps`-Arrays speichern, wenn eine Anwendung hinzugefügt wird. Fügen Sie die hervorgehobene Zeile in Ihre `add:`-Methode ein.

Persistence/HelloApplication38/BanishedApps.m

```
-(void)add:(NSRunningApplication *) app {
    if ([self contains:app]) return;
    [self.apps addObject:app.localizedName];
►   [self.apps writeToFile:self.dataFile atomically:YES];
}
```

Beeindruckend! Sie müssen das Array nur anweisen, sich selbst in einer Datei zu speichern, und den Pfad zu dieser Datei angeben. Arrays, Dictionaries, Strings, Zahlen, Datumsangaben und Daten wissen, wie sie sich selbst auf der Platte speichern und später wiederherstellen können.

Klicken Sie *Build & Run* an. Entfernen Sie einige der laufenden Anwendungen. Ich habe *loginwindow*, *AppleSpell.service*, *iChatAgent* und *iTunes Helper* entfernt. Sobald Sie eine Anwendung entfernt haben, wurde eine Plist erzeugt. Öffnen Sie die entsprechende Datei, und Sie sollten so etwas sehen wie das hier:

Key	Type	Value
▼ Root	Array	(4 items)
Item 0	String	loginwindow
Item 1	String	AppleSpell.service
Item 2	String	iChatAgent
Item 3	String	iTunes Helper

17.4 Eine Plist einlesen

Lassen Sie uns den anderen Weg gehen. Wir wollen die gespeicherten Informationen einlesen und das `apps`-Array in `BanishedApps` damit füllen. In der `init`-Methode stellen wir sicher, dass die *.plist*-Datei existiert. Ist das der Fall, initialisieren wir `apps` mit den gespeicherten Werten.

Persistence/HelloApplication38/BanishedApps.m

```
-(id)init {
   if (self =[super init]) {
      [self setSupportFile];
      if([[NSFileManager defaultManager] fileExistsAtPath:self.dataFile]){
          self.apps = [NSMutableArray arrayWithContentsOfFile:self.dataFile];
        } else {
          self.apps = [NSMutableArray arrayWithCapacity:5];
        }
    }
    return self;
}
```

Wir haben aber noch ein Problem: Entfernte Anwendungen werden immer noch augegeben, wenn *HelloApplication* gestartet wird. Um das zu verhindern, müssen wir `initWithNibName:bundle:` in `ActivityController.m` anpassen. Wir benutzen die schnelle Enumerierung, um jede Anwendung im `runningApplications`-Array des Workspaces zu überprüfen, und nehmen nur diejenigen in unser `runningApps`-Array auf, die nicht in der Liste stehen.

Persistence/HelloApplication38/ActivityController.m

```
-(id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)nibBundle{
    if (self =[super initWithNibName:nibName bundle:nibBundle] ) {
        self.banishedApps = [[BanishedApps alloc] init];
        self.runningApps = [NSMutableArray arrayWithCapacity:20];
►       for (NSRunningApplication *app in
►                [[NSWorkspace sharedWorkspace] runningApplications]) {
►           if (![self.banishedApps contains:app]) {
►                [self.runningApps addObject:app];
►           }
►       }
    }
    return self;
}
```

Klicken Sie auf *Build & Run*. Gelöschte Anwendungen sollten nicht im Tabellen-View erscheinen, selbst wenn Sie *HelloApplication* oder die gelöschten Anwendungen starten oder beenden.

17.5 Ein Archiv auf Festplatte speichern

Normalerweise verwenden Sie Plists zum Speichern relativ kleiner (meist konfigurationsbezogener) Mengen von Anwendungsdaten. Was aber tun, wenn Sie große Datenmengen oder einen Objektbaum festhalten wollen? In den meisten Fällen werden Sie die Techniken verwenden, die ich in Kapitel 22, *Core Data*, auf Seite 359 beschreibe. Es gibt aber immer wieder mal Fälle, in denen Sie die Daten in einem Archiv festhalten wollen.

Es gibt grundsätzlich zwei Ebenen, auf denen man einen Objektgraphen auf der Festplatte speichern kann. Auf der oberen Ebene wollen Sie ein Archiv aller zu sichernden (und später wieder abzurufenden) Objekte erzeugen. Diese Aktionen werden über Methoden von Instanzen der Klassen `NSKeyedArchiver` und `NSKeyedUnarchiver` übernommen. Sie übergeben das Objekt als Stamm des Baums, und jedes Objekt wird aufgefordert, sich selbst zu (de-)kodieren.

Auf dieser Objektebene muss jedes Objekt wissen, welche Teile seines Zustands über die Methode `encodeWithCoder:` gespeichert werden müssen. Das Laden der Daten ist die dazugehörige Umkehroperation und verwendet die `initWithCoder:`-Methode des Objekts.

Ich will hier keine neue Anwendung entwickeln, um dieses Konzept zu verdeutlichen. In den meisten Fällen sollten Sie entweder mit dem *.plist*-Ansatz zurechtkommen oder auf Core Data zurückgreifen. Doch nehmen wir einmal an, dass `apps` ein `NSMutableSet` anstelle eines `NSMutableArray` wäre.[2]

Nehmen wir an, Sie wollen den aktuellen Zustand Ihres `BanishedApps`-Objekts auf der Festplatte sichern. Es gibt nur zwei Variablen, über deren Speicherung wir nachdenken können: `apps` und `dataFile`. Es ergibt keinen Sinn, `dataFile` zu speichern, weil es sich nur um den Pfad auf das Verzeichnis handelt, in dem wir es speichern würden. Diesen Wert zu speichern, hätte für uns keinerlei Nutzen. Wenn es also daran geht, unser `BanishedApps`-Objekt zu speichern, müssen wir nur den Inhalt der `apps` entsprechend kodieren. Implementieren Sie eine passende `encodeWithCoder:`-Methode.

```
-(void) encodeWithCoder: (NSCoder *) coder {
     [coder encodeObject:self.apps forKey:@"bannedApps"];
}
```

2 Es ist einfach, ein `NSSet` in ein `NSArray` (und wieder zurück) umzuwandeln, weshalb Sie in der Praxis diese Umwandlung durchführen und bei der Plist bleiben würden. Ich will Ihnen hier nur zeigen, wie man einen NSCoder verwendet.

Sie rufen `encodeWithCoder:` nicht selbst auf. Es wird vom entsprechenden Archiver aufgerufen, wenn dieser den Objektbaum sichert. Unser `BanishedApps`-Objekt ist Teil des Baums, über dem ein `ActivityController`-Objekt liegt. Wir könnten also im Window-Controller ein Archiv mit dem folgenden Aufruf erzeugen:

```
[NSKeyedArchiver archiveRootObject:ac toFile:self.dataFile];
```

Hier ist `ac` die Instanz des `ActivityController`, der den Stamm unseres Baums bildet, und `dataFile` verweist auf die Datei, in der die Daten festgehalten werden.

Wir kehren den Prozess mit dem folgendem Aufruf um:

```
self.ac = [NSKeyedUnarchiver unarchiveObjectWithFile:self.dataFile];
```

Während des Dearchivierungsprozesses muss jedes Objekt wissen, wie es sich selbst dekodiert. Wir erledigen das in einer Variante der `init`-Methode namens `initWith-Coder:`. Innerhalb der `initWithCoder:`-Methode für `BanishedApps` könnte beispielsweise die folgende Zeile stehen:

```
self.apps = [coder decodeObjectForKey:@"bannedApps"];
```

Das sind die vier Schritte, die notwendig sind, wenn Sie Archive erzeugen wollen, deren Objekte aus Typen bestehen, die nicht wissen, wie sie sich selbst in eine Datei schreiben können. Nun kehren wir zu unserem Beispiel zurück und erlauben es dem Benutzer, Einstellungen an der Anwendung vorzunehmen.

17.6 Einstellungen lesen und verwenden

Wir wollen einige sehr einfache Einstellungen einfügen. Der Benutzer soll entscheiden können, ob er beim Start die gespeicherte Liste entfernter Anwendungen nutzen oder ganz neu anfangen will. In diesem Abschnitt gehen wir davon aus, das es diese Einstellung gibt und sie bereits gesetzt wurde.

Momentan fragen wir, ob die Datendatei existiert, bevor wir versuchen, `apps` von der Festplatte wiederherzustellen. Nun müssen wir bei der Überprüfung berücksichtigen, ob der Benutzer die Daten wiederherstellen wird und ob die Datendatei existiert.

Persistence/HelloApplication39/BanishedApps.m

```
-(id)init {
    if (self =[super init]) {
    [self setSupportFile];
►   if([self shouldLoadSavedRemovedApps] &&
         [[NSFileManager defaultManager] fileExistsAtPath:self.dataFile]){
          self.apps = [NSMutableArray arrayWithContentsOfFile:self.dataFile];
        } else {
          self.apps = [NSMutableArray arrayWithCapacity:5];
        }
    }
    return self;
}
```

Die Methode `shouldLoadSavedRemovedApps` liefert einen Wert zurück, der auf den Einstellungen basiert.

Persistence/HelloApplication39/BanishedApps.m

```
-(BOOL) shouldLoadSavedRemovedApps {
    return[[NSUserDefaults standardUserDefaults]
                           boolForKey:@"LoadSavedRemovedApps"];
}
```

Dieses Muster ist uns vertraut. Sie fragen die Standardbenutzervorgaben in der gleichen Weise ab, wie Sie den Standard-Workspace oder den Dateimanager abfragen. Sobald Sie die benötigte Instanz besitzen, fragen Sie von ihr denjenigen Wert ab, der dem Schlüssel `LoadSavedRemovedApps` entspricht, und geben diesen Wert als `BOOL` zurück.

17.7 Die „Werkseinstellungen“ festlegen

Lassen sie uns einen Standardwert für `shouldLoadSavedRemovedApps` festlegen.

Es gibt fünf Ebenen von Voreinstellungen, die Sie in einer Cocoa-Anwendung festlegen können. Wir wollen uns nur mit zweien davon beschäftigen: den Werkseinstellungen und den Benutzereinstellungen. Wir ignorieren über die Kommandozeile übergebene Einstellungen, für das gesamte System geltende Einstellungen und von der Lokalisierung abhängige Einstellungen. Wir werden zuerst die Werkseinstellungen laden und dann überprüfen, ob der Benutzer die Einstellungen verändert hat.

In diesem Abschnitt sehen wir uns die Voreinstellungen in der Registrierungsdomain an. Sie können das als die Werkseinstellungen betrachten – die Werte, bevor der Benutzer Anpassungen vornimmt. Lassen Sie uns den Wert von `should-LoadSavedRemovedApps` auf `NO` setzen.

Der beste Ort zur Festlegung der Registrierungsdomain-Voreinstellungen ist die `initialize`-Methode. Der Grund dafür ist die Tatsache, dass die `initialize`-Methode eine Klassenmethode ist, die vor allen Methoden in `HelloApplicationAppDelegate` oder einer Subklasse aufgerufen wird. Alle Anweisungen in der `initialize`-Methode werden vor den Anweisungen in unserem App-Delegate ausgeführt – selbst die Anweisungen in der `applicationDidFinishLaunching:`-Methode.

Lassen Sie uns die `initialize`-Methode in unserem App-Delegate anlegen:

Persistence/HelloApplication39/HelloApplicationAppDelegate.m

```
+(void)initialize {
   NSDictionary *defaults =
     [NSDictionary dictionaryWithObject:[NSNumber numberWithBool:NO]
                                 forKey:@"LoadSavedRemovedApps"];
     [[NSUserDefaults standardUserDefaults] registerDefaults:defaults];
}
```

Klicken Sie *Build & Run* an. Bei jedem Start der Anwendung beginnen Sie mit einem leeren Array gelöschter Anwendungen. Ändern Sie das `NO` in ein `YES`, und das Programm startet mit der gespeicherten Liste gelöschter Anwendungen.

Tatsächlich müssen Sie nicht so konkret werden, wenn Sie den `BOOL`-Wert im Dictionary ablegen. Sie können beim Lesen der Voreinstellungen den Wert `YES` oder `NO` auch als String übergeben und daraus einen `BOOL`-Wert erzeugen.

Persistence/HelloApplication40/HelloApplicationAppDelegate.m

```
+(void)initialize {
   NSDictionary *defaults =
     [NSDictionary dictionaryWithObject:@"NO"
                                 forKey:@"LoadSavedRemovedApps"];
     [[NSUserDefaults standardUserDefaults] registerDefaults:defaults];
}
```

Ein typisches Beispiel für die Speicherung eines etwas komplexeren Objekts in der Voreinstellungsdatei ist das Speichern einer Farbe. In unserer Anwendung gibt es dafür keinen Bedarf, doch lassen Sie uns kurz sehen, wie man das machen würde. Sie müssen die Daten des Farbobjekts abspeichern. Dazu nutzen Sie die Technik zur Erzeugung eines Archivs, die Sie an anderer Stelle in diesem Kapitel kennengelernt haben.

Nehmen Sie die Änderungen nicht an unserem Projekt vor – ich will Ihnen nur zeigen, wie es funktioniert.

Um eine Farbe in den Einstellungen zu speichern, erzeuge ich ein neues Farbobjekt namens `myColor` und setze seinen Wert beispielsweise auf rot. Dann verwende ich einen `NSArchiver`, um die Farbe in ein Datenobjekt umzuwandeln. Alles andere ist wie gehabt. Tragen Sie die Daten in ein Dictionary ein und registrieren Sie Ihre Voreinstellungen.

```
+(void)initialize{
  NSColor *myColor = [NSColor redColor];
  NSData *colorData=[NSArchiver archivedDataWithRootObject:myColor];
  NSDictionary *defaults = [NSDictionary dictionaryWithObject:colorData
                                                       forKey:@"BackgroundColor"];
  [[NSUserDefaults standardUserDefaults] registerDefaults:defaults];
}
```

Um den Prozess umzukehren, ziehen wir ein Datenobjekt aus den Standard-Voreinstellungen und verwenden einen `NSUnarchiver`, um die Daten in ein Farbobjekt umzuwandeln.

```
NSData *colorData = [[NSUserDefaults standardUserDefaults]
                                        dataForKey:@"BackgroundColor"];
NSColor *myBackgroundColor =
        (NSColor *)[NSUnarchiver unarchiveObjectWithData:colorData];
```

Sie können diese Technik benutzen, um jeden Nichtstandard-Typ in einer Einstellungsdatei zu speichern und wieder abzurufen. Beachten Sie, dass wir die Standard-Benutzereinstellungen an jeder Stelle der Anwendung lesen können. Die Objekte, die die Einstellungen lesen und schreiben, müssen nichts voneinander wissen.

17.8 Das Setzen der Benutzereinstellungen vorbereiten

Wir wollen die Anwendung so konfigurieren, dass sie Einstellungen an der richtigen Stelle speichert. Dateien mit Benutzereinstellungen für Ihre Anwendung sollen unter `~/Library/Preferences` in einer Datei namens `com.ihrunternehmen.namederanwendung.plist` stehen. In unserem Fall nennen wir die Datei `com.pragprog.HelloApplication.plist`.

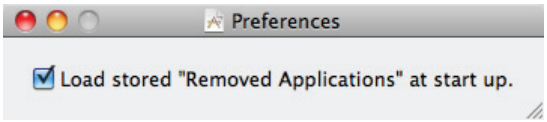
Sie können das in der Datei `HelloApplication-Info.plist` einstelllen. Sie finden diese Datei unter *Resources* im Abschnitt *Groups & Files* Ihres Projektfensters. Setzen Sie den Bundle-Identier auf den Wert `com.pragprog.HelloApplication` und speichern Sie Ihre Änderungen.

Key	Value
▼ Information Property List	(13 items)
Localization native development re	English
Executable file	${EXECUTABLE_NAME}
Icon file	
Bundle identifier	com.pragprog.HelloApplication
InfoDictionary version	6.0
Bundle name	${PRODUCT_NAME}
Bundle OS Type code	APPL
Bundle creator OS Type code	????
Bundle versions string, short	1.0
Minimum system version	${MACOSX_DEPLOYMENT_TARGET}
Bundle version	1
Main nib file base name	MainMenu
Principal class	NSApplication

17.9 Das Nib für das Einstellungsfenster

Nun wollen wir die *.nib*-Datei erzeugen, die wir für unser Einstellungsfenster verwenden wollen. Legen Sie eine neue Datei im Ordner `Resources` an. Wählen Sie die Vorlage `Mac OS X > User Interface > Window XIB`. Sichern Sie die Datei unter dem Namen `Preferences.xib`.

Klicken Sie `Preferences.xib` doppelt an, um den Interface Builder zu öffnen. Deaktivieren Sie die Checkbox *Visible At Launch* und ändern Sie den Titel des Fensters in *Preferences* (also Einstellungen). Ziehen Sie eine Checkbox hinein und fügen Sie einen beschreibenden Text hinzu.



Eine schöne Sache bei der Entwicklung all dieser kleinen Nibs besteht darin, dass unser Weg zum Ziel viel klarer wird. Als wir Nibs mit einem View aufbauten, wussten wir, dass ein View-Controller zu entwickeln und als *File's Owner* festzulegen war. Wenn das Nib nicht viel mehr als ein Fenster enthält, neigen wir dazu, einen Window-Controller zu entwickeln und diesen als *File's Owner* anzugeben.

Legen Sie eine neue Klasse namens `PreferencesController` an. Nutzen Sie dazu die `NSWindowController`-Vorlage. Fügen Sie ein Outlet in die Header-Datei ein, um die Verbindung mit der Checkbox herzustellen. Fügen Sie außerdem eine Aktion ein, die aufgerufen wird, wenn der Benutzer die Checkbox aktiviert bzw. deaktiviert.

Persistence/HelloApplication40/PreferencesController.h

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

@interface PreferencesController : NSWindowController {
    NSButton *loadSavedRemovedAppsCheckbox;
}
@property IBOutlet NSButton *loadSavedRemovedAppsCheckbox;
-(IBAction) toggleLoadSavedRemovedApps: (id)sender;

@end
```

Der `PreferencesController` hat nicht viel zu tun. Wenn das Fenster mit den Einstellungen erscheint, wollen wir die Checkbox auf den Zustand setzen, der in den Benutzereinstellungen gespeichert ist:

Persistence/HelloApplication40/PreferencesController.m

```
-(void) awakeFromNib {
     [self.loadSavedRemovedAppsCheckbox
            setState:[[NSUserDefaults standardUserDefaults]
        boolForKey:@"LoadSavedRemovedApps"]];
}
```

Und wenn der Benutzer die Checkbox (de-)aktiviert, wollen wir den neuen Wert in den Benutzereinstellungen speichern:

Persistence/HelloApplication40/PreferencesController.m

```
-(IBAction) toggleLoadSavedRemovedApps: (id) sender {
     [[NSUserDefaults standardUserDefaults]
            setBool:[self.loadSavedRemovedAppsCheckbox state]
                forKey:@"LoadSavedRemovedApps"];
}
```

Wir verbinden den `PreferencesController` mit dem von ihm verwalteten Nib. Klicken Sie `Preferences.xib` doppelt an. Legen Sie die *File's Owner*-Klasse mit dem `PreferencesController` fest. Verbinden Sie dessen `window`-Outlet mit dem `Window`. Verbinden Sie dessen `loadSavedRemovedAppsCheckbox`-Outlet und die `toggleLoadSavedRemovedApps`-Aktion mit der `Checkbox`. Wählen Sie das `Window` und verbinden Sie dessen `delegate` mit dem *File's Owner*.

Speichern Sie Ihre Änderungen. Wir haben das Einstellungsfenster und seinen Controller erzeugt. Wir haben den Controller implementiert und alles mit dem neuen Nib verbunden. Gleichzeitig haben wir den `ActivityController` so eingestellt, dass er diese Einstellungen auch verwendet. Nun müssen wir nur noch das Einstellungsfenster öffnen, wenn der Benutzer `HelloApplication` > `Preferences` anklickt.

17.10 Das Einstellungsfenster aktivieren

Der letzte Schritt besteht darin, das Einstellungsfenster in unsere Anwendung zu integrieren. Eine einfache Lösung besteht darin, eine Aktion namens `openPreferences:` in die `HelloApplicationAppDelegate`-Klasse einzufügen. Diese wird aufgerufen, wenn der Benutzer das Einstellungsmenü wählt. Wir fügen auch eine Eigenschaft für unseren *PreferencesController* hinzu.

Persistence/HelloApplication40/HelloApplicationAppDelegate.h

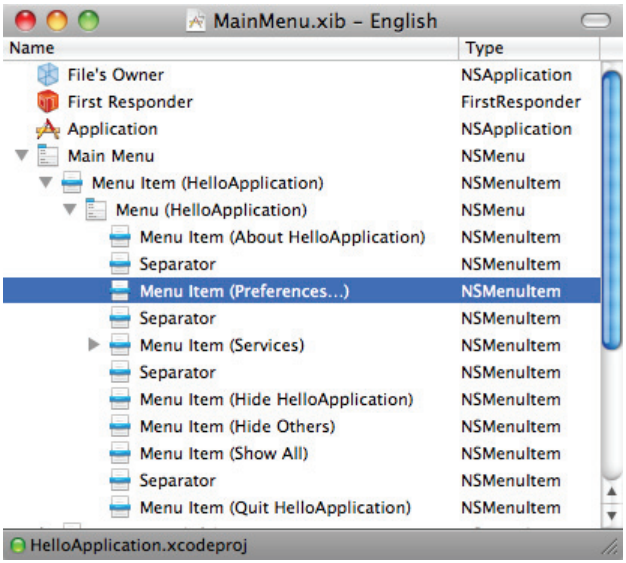
```
#import <Cocoa/Cocoa.h>

@interface HelloApplicationAppDelegate:NSObject<NSApplicationDelegate> {
}
-(IBAction) openPreferences:(id) sender;
@end
```

Die Methode `openPreferences:` initialisiert den `PreferenceController`, lädt die *.nib*-Datei und zeigt das Fenster mit `showWindow:` an.

Persistence/HelloApplication40/HelloApplicationAppDelegate.m

```
-(IBAction) openPreferences: (id)sender {
    PreferencesController *prefController = [[PreferencesController alloc]
                          initWithWindowNibName:@"Preferences"];

 [prefController showWindow:self];
}
```

Klicken Sie `MainMenu.xib` doppelt an und suchen Sie sich den Menüpunkt *Preferences...* heraus.



Verwenden Sie den Connections Inspector, um die `HelloApplicationDelegate`-Aktion `openPreference:` mit dem Menüpunkt *Preferences...* zu verknüpfen.

Wir initialisieren also den `PreferencesController` und laden die *.nib*-Datei `Preferences`, wenn der Benutzer die *Einstellungen...* anklickt. Wir müssen die Ressourcen noch aufräumen, wenn der Benutzer die Einstellungen beendet. Das Fenster muss nicht existieren, wenn es nicht sichtbar ist, weshalb wir es freigeben und ein neues erzeugen, wenn der Benutzer die Einstellungen später noch einmal verändern will.

Der `PreferencesController` ist der Delegate für das Fenster, weshalb wir die folgende Delegate-Methode implementieren können:

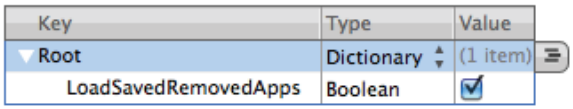
Persistence/HelloApplication40/PreferencesController.m

```
-(void)windowWillClose: (NSNotification *) notification {
     [self autorelease];
}
```

Das ist ein weiterer Vorteil vieler kleiner, spezialisierter Nibs: Wir können Ressourcen bei Bedarf laden und sie danach wieder freigeben.

Wenn Sie an diesem Punkt *Build & Run* anklicken, können Sie das Einstellungsfenster entweder über das Menü oder über das Tastaturkürzel ⌘ erreichen. Alles ist vorbereitet, damit Sie Einstellungen festlegen und abrufen können. Spielen Sie ein wenig herum, indem Sie die Einstellungen verändern, das Programm beenden und wieder neu starten.

Sie können sogar die Datei mit den Einstellungen öffnen (also die Plist) und sich die gesetzten Werte ansehen.



In diesem Kapitel haben Sie gelernt, wie man kleine Datenmengen speichert. Sie haben gesehen, wie man Plists aus einfachen Typen wie `NSArray` erzeugt und man komplexere Daten archiviert. Sie haben auch gesehen, wie man mit Benutzereinstellungen umgeht. Wenn wir größere Datenmengen oder komplexere Daten wollen, halten wir uns an Core Data. Im nächsten Kapitel werden wir verschiedene Views verwenden und nähern uns dann vorsichtig Core Data an.

Kapitel 18

Views wechseln

In Kapitel 16, *Daten in einer Tabelle darstellen*, auf Seite 259 haben wir es dem Windows-Controller leicht gemacht, zwischen View-Controller und View zu wechseln. Die gute Nachricht ist, dass Sie den View ganz leicht wechseln können, indem Sie den Namen der View-Controller-Klasse und den Namen des Nib ändern und die Sache neu kompilieren. Die schlechte Nachricht ist, dass der Benutzer das nicht zur Laufzeit tun kann.

In diesem Kapitel werden wir eine neue Einstellung entwickeln, mit deren Hilfe der Benutzer festlegen kann, welcher View beim Start der Anwendung verwendet werden soll. Wir werden diese Einstellung dann nutzen, um den View darzustellen, den der Benutzer gewählt hat. Danach werden wir das Beispiel so erweitern, dass der Benutzer während der laufenden Anwendungen zwischen den Views wechseln kann, sooft er will.

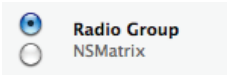
Wir haben den ersten Akt unserer Geschichte zusammengefasst, indem wir unseren einfachen Browser für das iPhone neu geschrieben haben. Dabei gab es nicht viel neues Material; vielmehr es ging darum, das Gelernte in einem neuen Kontext zu vertiefen.

Nachdem der Vorhang nach dem zweiten Akt gefallen ist, wollen wir uns einige der erlernten Techniken noch einmal ansehen, während Sie an diesem Beispiel arbeiten. Zwar werden Sie hier und da etwas Neues lernen, aber das Hauptziel besteht darin, Ihnen die Vorteile der Arbeit mit mehreren Nibs und kleinen, wohldefinierten Klassen und Methoden zu verdeutlichen.

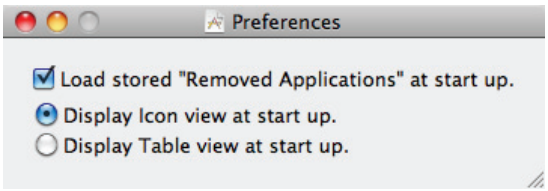
18.1 Mit Radiobuttons arbeiten

Lassen Sie uns eine Reihe von Radiobuttons in die Einstellungen einfügen. Klicken Sie `Preferences.xib` doppelt an und suchen Sie in der Library nach *Radio*.

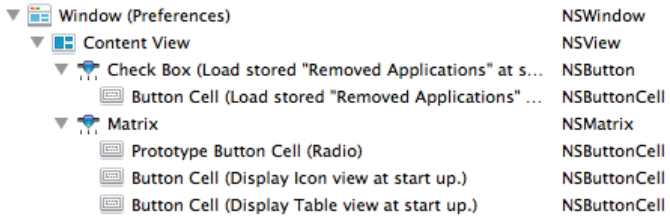
Sie sollten dieses Widget hier finden:



Ziehen Sie es in das Einstellungsfenster. Passen Sie die Größe des Widgets und des Fensters an und fügen Sie rechts neben den beiden Radiobuttons Text ein, sodass die Sache etwa so aussieht:



Sehen Sie sich die Hierarchie der GUI-Elemente im `Preferences.xib`-Dokumentenfenster an.



Wir könnten jedes `NSButtonCell`-Objekt der Radiobutton-Gruppe individuell verknüpfen und einzeln verarbeiten. Wir wollen die Gruppe aber als Ganzes mithilfe des `NSMatrix`-Objekts verarbeiten.

Um den gewählten Radiobutton aus dem `PreferencesController`-Objekt heraus zu setzen, fügen Sie eine Eigenschaft namens `viewGroup` vom Typ `NSMatrix` in die `PreferencesController`-Klasse ein. Wir müssen außerdem eine Aktion in den `PreferencesController` einfügen, die immer dann aufgerufen wird, wenn ein Radiobutton gewählt wird. Nennen Sie die Aktion `chooseView:` und sichern Sie `PreferencesController.h`. Im IB verwenden Sie den Connections Inspector, um Outlet und Aktion mit `NSMatrix` zu verknüpfen.

Wenn wir den Controller mit der `NSMatrix` und nicht mit den einzelnen Buttons verknüpfen, müssen wir die beiden Buttons auseinanderhalten können, damit wir wissen, welcher gewählt wurde. Bei der Arbeit mit Tabellen haben wir den Spaltenbezeichner benutzt. `NSButtonCells` kennen keine Bezeichner, aber dafür Tags.

Im Interface Builder wählen Sie den Button, den Sie mit „Display Icon view at startup“ bezeichnet haben. Sehen Sie sich den unteren Rand des Attributes Inspector an, und Sie erkennen das Attribut `Tag` mit dem Wert `1`. Das `Tag`-Attribut[1] für die andere Button-Zelle weist entsprechend den Wert `0` auf.

18.2 Einstellungen für den Start-View einfügen

Wir müssen vier Dinge erledigen, damit diese Einstellungen funktionieren:

1. Einen Namen und ein Format für die Einstellung wählen.
2. Einen Standardwert festlegen, der verwendet wird, bevor der Benutzer seine eigenen Einstellungen vornimmt.
3. Die Einstellungen einlesen, wenn unser `PreferencesController` aus dem Nib geladen wird.
4. Die Einstellungen speichern, wenn der Benutzer einen Radiobutton wählt.

Wir könnten unsere Einstellungen als booleschen Wert abspeichern, wie wir das für die Checkbox getan haben. Das würde für's Erste funktionieren, da wir nur mit zwei Views arbeiten. Da wir das bereits in dieser Form implementiert haben, wollen wir es hier anders lösen.

Wir wollen den Tag des momentan gewählten Buttons als Wert der Einstellung speichern. Der Tag ist vom Typ `int`. Wir müssen ihn beim Schreiben oder Lesen der Einstellungen also in eine `NSNumber` (bzw. wieder zurück) konvertieren. Wir erzeugen eine `NSNumber` aus einem `int`-Wert mithilfe der Klassenmethode `numberWithInt:`. Andersherum extrahieren wir den `int`-Wert aus `NSNumber` mit der Methode `intValue`.

Nachdem wir nun ein Format für unsere Einstellung besitzen, können wir es so benennen, dass wir wissen, um was es geht. Wir wollen es *TagForView* nennen. Wir können den Standardwert in `HelloApplicationAppDelegate.m` direkt unterhalb des Standardwerts für `LoadSavedRemovedApps` angeben.

1 Wenn diese Tags andere Werte aufweisen, ändern Sie sie bitte in `1` und `0` ab.

ChangingViews/HelloApplication41/HelloApplicationAppDelegate.m

```
+(void)initialize {
    NSDictionary *defaults =
        [NSDictionary dictionaryWithObjectsAndKeys:
        @"NO",@"LoadSavedRemovedApps",
        [NSNumber numberWithInt:1], @"TagForView",
        nil];
    [[NSUserDefaults standardUserDefaults] registerDefaults:defaults];
}
```

Im vorigen Kapitel haben wir die Checkbox entsprechend dem in den Einstellungen gespeicherten Wert aktiviert bzw. deaktiviert. Das machen wir auch für die Radiobuttons. Ziehen Sie die `NSNumber` aus den Einstellungen heraus und wandeln Sie sie in ein `int` um. Weisen Sie das `NSMatrix`-Objekt an, die Zelle mit diesem Tag auszuwählen.

ChangingViews/HelloApplication41/PreferencesController.m

```
-(void) awakeFromNib {
    [self.loadSavedRemovedAppsCheckbox
                        setState:[[NSUserDefaults standardUserDefaults]
                                boolForKey:@"LoadSavedRemovedApps"]];
    [self.viewGroup selectCellWithTag:
                        [[[NSUserDefaults standardUserDefaults]
                                        objectForKey:@"TagForView"] intValue]];
}
```

Die Aktion `chooseView:` wird immer dann aufgerufen, wenn der Benutzer einen Radiobutton auswählt. Wir nehmen uns den `tag` der vom Benutzer gewählten Zelle, wandeln ihn in eine `NSNumber` um und aktualisieren den Wert der `TagForView`-Einstellung.

ChangingViews/HelloApplication41/PreferencesController.m

ChangingViews/HelloApplication41/PreferencesController.m

```
-(IBAction) chooseView: (id) sender {
    [[NSUserDefaults standardUserDefaults]
       setObject:[NSNumber numberWithInt:[[sender selectedCell] tag]]
           forKey:@"TagForView"];
}
```

Klicken Sie *Build & Run* an. Sie können nun auswählen, welcher View beim Programmstart geladen wird. Schließen Sie das Einstellungsfenster und öffnen Sie es erneut. Beenden Sie die Anwendung und starten Sie sie neu. Egal, was Sie tun – die Einstellung wird immer gespeichert, sobald sie gesetzt wird. Nun wollen wir diese Einstellung benutzen, um den richtigen View zu laden.

18.3 Übung: Den richtigen View laden

Passen Sie `MyWindowController.m` so an, dass er den Wert der `TagForView`-Einstellung einliest. Das Programm sollte mit dem Icon-View starten, wenn der Wert 1 ist, bzw. mit dem Tabellen-View, wenn der Wert `0` ist.

18.4 Übung: Den richtigen View laden

Wir haben den View-Controller und den Nib-Namen in den ersten beiden Zeilen von `setUpView` festgelegt. Diese Zeilen werden wir nun in die Methoden `loadIconView` und `loadTableView` auslagern. Diese Methoden wollen wir dann aufrufen, um den gewünschten View zu laden.

ChangingViews/HelloApplication42/MyWindowController.m

```
-(void) setUpView {
    self.currentApp = [[CurrentApp alloc] init];
    self.currentApp.delegate = self.ac;
     [self.window setContentSize:[self.ac.view bounds].size];
     self.window.contentView = self.ac.view;
     [self.ac applicationDidLaunch:
                         [NSRunningApplication currentApplication]];
}
-(void) loadIconView {
    self.ac = [[IconViewController alloc]
                initWithNibName:@"IconView" bundle:nil];
     [self setUpView];
}
-(void) loadTableView {
    self.ac = [[ActivityController alloc]
                initWithNibName:@"ActivityView" bundle:nil];
     [self setUpView];
}
```

Legen Sie eine Methode namens `shouldLoadIconView` an, um den in den Einstellungen gespeicherten Wert einzulesen:

ChangingViews/HelloApplication42/MyWindowController.m

```
-(BOOL) shouldLoadIconView{
    return (1 == [[[NSUserDefaults standardUserDefaults]
                        objectForKey:@"TagForView"] intValue]);
}
```

Die Methode `shouldLoadIconView` ist kurz, und Sie könnten versucht sein, ohne sie auszukommen. Ich habe sie eingeführt, um die Logik in `initWithWindowNibName:` verständlicher zu machen. Das ist einfacher zu lesen als die direkte Abfrage der Einstellungen.

ChangingViews/HelloApplication42/MyWindowController.m

```
-(id) initWithWindowNibName:(NSString *)windowNibName {
    if (self =[super initWithWindowNibName:windowNibName]) {
        if ([self shouldLoadIconView])[self loadIconView];
        else [self loadTableView];
        [self showWindow:self];
    }
    return self;
}
@end
```

Klicken Sie *Build & Run* an, und Ihre Anwendung wird mit dem gewünschten View gestartet. Der Rest des Kapitels widmet sich unserem Ziel, den View jederzeit wechseln zu können.

18.5 „Magic Numbers“ eliminieren

Ich möchte für diejenigen einen kleinen Abstecher machen, denen die Verwendung von „Magic Numbers“ in unserer Lösung nicht gefällt. Legen Sie eine Kopie unseres Projekts an, da wir im nächsten Abschnitt wieder mit diesem Stand fortfahren. Sie können diesen Abschnitt aber auch problemlos überspringen und den Faden im nächsten Abschnitt wieder aufnehmen.

Das Problem ist hier die Verwendung der Zahl 1:

ChangingViews/HelloApplication42/MyWindowController.m

```
-(BOOL) shouldLoadIconView{
    return (1 == [[[NSUserDefaults standardUserDefaults]
                   objectForKey:@"TagForView"] intValue]);
}
```

Die 1 ist eine „magische Zahl“, die an den Tag unserer Radiobuttons gebunden ist, die wir zur Auswahl des zu ladenden Views verwenden. Sie könnten `#define ICON_VIEW_ID 1` gleich unter `@implementation` angeben und `ICON_VIEW_ID` anstelle der 1 verwenden. Das macht den Zweck des Codes für andere Entwickler verständlicher. Sie können auch eine öffentliche Konstante im Application-Delegate deklarieren und definieren.

Lassen Sie uns eine Plist namens *Views* erzeugen und verwenden:

ChangingViews/HelloApplication42alt/Views

```
<?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>
```

```
        <dict>
                <key>class</key>
                <string>ActivityController</string>
                <key>nib</key>
                <string>ActivityView</string>
        </dict>
        <dict>
                <key>class</key>
                <string>IconViewController</string>
                <key>nib</key>
                <string>IconView</string>
        </dict>
</array>
</plist>
```

Das ist kurz genug, um es noch von Hand einzugeben. Es handelt sich um ein Array von Dictionaries. Das Dictionary an Index `0` repräsentiert den Tabellen-View, und das Dictionary an `1` den Icon-View. Wenn wir weitere Views hinzufügen, fügen wir einfach weitere Einträge in das Array ein. Jedes Dictionary besitzt einen String mit dem Schlüssel „class“. Der dazugehörige Wert ist der Klassenname des Views.

Wenn Sie diese Datei nicht von Hand schreiben wollen, können Sie das auch programmtechnisch erledigen lassen. Fügen Sie Folgendes in die `applicationDidFinishLaunching:`-Methode Ihres App-Delegates ein, klicken Sie auf *Build & Run* und entfnern Sie den Code wieder. Das erzeugt eine Datei namens `Views` in ihrem *build*-Verzeichnis.

```
NSDictionary *tableView = [NSDictionary dictionaryWithObjectsAndKeys:
    @"ActivityView",@"nib",@"ActivityController",@"class", nil];
NSDictionary *iconView = [NSDictionary dictionaryWithObjectsAndKeys:
    @"IconView",@"nib",@"IconViewController",@"class", nil];
NSArray *viewArray =
     [NSArray arrayWithObjects:tableView, iconView, nil];
[viewArray writeToFile:@"Views" atomically:YES];
```

Wie auch immer Sie die `Views`-Datei erzeugen, Sie müssen sie in das Projekt einbinden. Öffnen Sie mit einem Control-Klick den `Folder` unter *Groups & Files* und wählen Sie `Add > Existing Files...` Bewegen Sie sich zu Ihrer `Views`-Datei und klicken Sie den *Add*-Button an. Wenn die Drop-down-Liste erscheint, aktivieren Sie die Checkbox bei *Copy items into destination group's folder (if needed)* und geben den *Reference Type* im Drop-down-Menü mit *Relative to Project* an. Das ermöglicht uns den Zugriff auf diese Datei, ohne den langen Pfad angeben zu müssen. Klicken Sie zum Schluss den *Add*-Button an.

Bis jetzt haben wir unseren Code nicht verändert. Wir haben nur die magischen Zahlen und die Entscheidungsfindung in die neu erzeugte Konfigurationsdatei verlagert. Nun wollen wir diese in `MyWindowController.m` nutzen. Wir ersetzen sowohl die `loadIconView`- als auch die `loadTableView`-Methode durch die folgende:

ChangingViews/HelloApplication42alt/MyWindowController.m

```
-(void)loadView {
    NSArray *viewArray = [NSArray arrayWithContentsOfFile:
                    [[NSBundle mainBundle] pathForResource:@"Views"
                                                    ofType:nil]];
    NSDictionary *view = [viewArray objectAtIndex:
                    [[[NSUserDefaults standardUserDefaults]
                                objectForKey:@"TagForView"] intValue]];
    self.ac =
        [[NSClassFromString([view objectForKey:@"class"]) alloc]
        initWithNibName:[view objectForKey:@"nib"] bundle:nil];
    [self setUpView];
}
```

Wir beginnen damit, unser Array von Dictionaries aus der `Views`-Datei zu erzeugen. Dann benutzen wir den Wert der `TagForView`-Einstellung, um das gewünschte Dictionary auszuwählen. Dann verwenden wir die im Dictionary gespeicherten Klassen- und Nib-Namen, um einen neuen View-Controller zu erzeugen.

Wir können in der `initWithWindowNibName`-Methode die `shouldLoadIconView`-Methode löschen und die `if`-Anweisung entfernen:

ChangingViews/HelloApplication42alt/MyWindowController.m

```
-(id) initWithWindowNibName:(NSString *)windowNibName {
    if (self =[super initWithWindowNibName:windowNibName]) {
►         [self loadView];
          [self showWindow:self];
    }
    return self;
}
@end
```

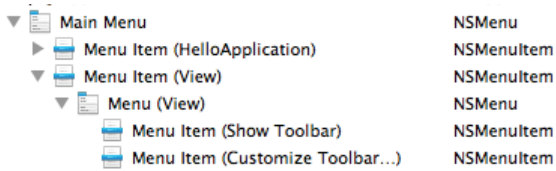
Klicken Sie *Build & Run* an. Die Anwendung läuft wie bisher auch. Wir haben die Magic Number und die Entscheidungsfindung eliminiert. Ich bin allerdings nicht sicher, ob dieser Code leichter zu verstehen ist. Ich werde mit der Version weitermachen, die wir vor diesem Abschnitt genutzt haben.[2]

2 Sie können für den Rest dieses Kapitels mit diesem Code weiterarbeiten. Ich habe nicht alle notwendigen Änderungen beschrieben, doch mein Mathematiker-Hintergrund lässt mich an dieser Stelle sagen: „Das ist dem Leser als Übung überlassen.“

18.6 Die Menüleiste anpassen

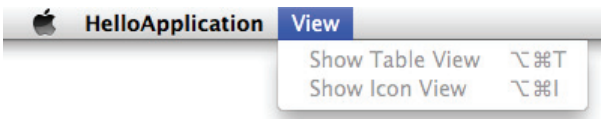
Wenn Sie die Anwendung ausführen, sehen Sie eine Menüleiste, die mit Elementen angefüllt ist, die für den Benutzer größtenteils irrelevant sind. Es gibt in den `File`-, `Edit`- und `Format`-Menüs nichts, was irgendjemand benötigen könnte. Lassen Sie uns also ein wenig aufräumen.

Klicken Sie `MainMenu.xib` doppelt an. Im Dokumentenfenster wählen und löschen Sie nacheinander die `NSMenuItems` namens `File`, `Edit`, `Format`, `Window` und `Help`.[3] Ihr Hauptmenü sollte im Dokumentenfenster wie folgt aussehen:



Ändern Sie die beiden unteren Elemente. Wählen Sie `Show Toolbar` und öffnen Sie den Attributes Inspector. Ändern Sie den Titel in `Show Table View` ab und belassen Sie das Tastaturkürzel bei ⌥⌘T (`Option-Command-T`). Wählen Sie ebenso den Menü-Eintrag `Customize Toolbar...` und ändern sie seinen Titel in Show Icon View ab. Setzen Sie das Tastaturkürzel auf ⌥⌘I (`Option-Command-I`).

Sichern Sie Ihre Änderungen und klicken Sie auf *Build & Run*. Unsere neue Menüleiste sieht gut aus:



Die einzelnen Elemente und ihre Tastaturkürzel sind da. Sie sind ausgegraut, weil sie vom Benutzer momentan noch nicht ausgewählt werden können. Sie machen im Moment noch nichts.

18.7 Das Hauptfenster verschieben

Wenn ein Benutzer einen der beiden Menüpunkte auswählt, muss natürlich der Window-Controller reagieren. Der Window-Controller ist so eingerichtet, dass er einen View durch den anderen ersetzt.

3 Wir hätten das `Window`-Menü für den `Minimize`-Befehl beibehalten können, doch wir wollen uns auf das `View`-Menü konzentrieren.

Unglücklicherweise liegen die Menüelemente und der Window-Controller in separaten Nibs, wir können also im Window-Controller nicht einfach eine Aktion anlegen und mit dem Menüpunkt verknüpfen.

Öffnen Sie `MainMenu.xib` im Interface Builder. Ziehen Sie eine Instanz von `MyWindow-Controller` aus dem *Classes*-Reiter der Library in das Dokumentenfenster. Nun müssen wir festlegen, wie unser Window-Controller und das `MainWindow`-Nib zu laden sind.

Die Methode `applicationDidFinishLaunching:` in `HelloApplication-AppDelegate` enthält eine einzige Zeile:

```
[[MyWindowController alloc] initWithWindowNibName:@"MainWindow"];
```

Diese müssen wir entfernen. Indem wir `MyWindowController` in das `MainMenu` ziehen, instanziieren wir es auch an dieser Stelle. Wir dürfen keine separate Instanz im App-Delegate erzeugen, da wir dann eine Instanz betrachten, während wir die Nachrichten an eine andere senden. Löschen Sie die Zeile, damit die `applicationDidFinishLaunching:`-Methode leer ist.

Wenn Sie an diesem Punkt *Build & Run* anklicken, startet die Anwendung zwar, aber wir sehen kein Fenster. Sehen Sie sich `MyWindowController.m` an: Es gibt eine Methode namens `initWithWindowNibName:`, aber dort steht nichts drin, was ihr den Namen der *.nib*-Datei angeben würde.

Fügen Sie die folgende Implementierung von `init` direkt unter `initWithWindowNibName:` ein:

ChangingViews/HelloApplication43/MyWindowController.m

```
-(id) init {
    return [self initWithWindowNibName:@"MainWindow"];
}

@end
```

Dieser einfache Aufruf gibt uns alles, was wir brauchen. Er besagt, dass bei der Erzeugung einer neuen `MyWindowController`-Instanz die *.nib*-Datei, die das Fenster enthält, `MainWindow` heißt.

Klicken Sie auf *Build & Run*, und das Fenster ist nach dem Start wieder sichtbar.

18.8 Übung: Views wechseln (weitgehend)

Machen Sie Aktionen aus den Methoden `loadIconView` und `loadTableView` in `MyWindowController.m`, und verbinden Sie diese mit Ihren Menüpunkten. Unsere Anwendung sollte weitgehend funktionieren. Ich sage „weitgehend“, weil jedesmal *HelloApplication* als erste Anwendung erscheint, wenn Sie zurück in den Icon-View wechseln. Das wollen wir als Nächstes korrigieren.

18.9 Lösung: Views wechseln (weitgehend)

Fügen Sie Deklarationen für die beiden Aktionen in `MyWindowController.h`: ein:

ChangingViews/HelloApplication44/MyWindowController.h

```
#import <Cocoa/Cocoa.h>
#import "ActivityMonitorDelegate.h"
@class CurrentApp;

@interface MyWindowController : NSWindowController {
    CurrentApp *currentApp;
    NSViewController <ActivityMonitorDelegate> *ac;
}
@property CurrentApp *currentApp;
@property NSViewController <ActivityMonitorDelegate> *ac;
► -(IBAction) loadIconView:(id) sender;
► -(IBAction) loadTableView: (id) sender;
@end
```

Nehmen Sie die dazu gehörigen Änderungen der Methodensignaturen in der Implementierungsdatei vor.

ChangingViews/HelloApplication44/MyWindowController.m

```
-(IBAction) loadIconView:(id) sender {
►     self.ac = [[IconViewController alloc]
                initWithNibName:@"IconView" bundle:nil];
    [self setUpView];
}
► -(IBAction) loadTableView: (id) sender {
    self.ac = [[ActivityController alloc]
                initWithNibName:@"ActivityView" bundle:nil];
    [self setUpView];
}
```

Sie müssen auch `self` als Parameter hinzufügen, wenn Sie diese Methoden in `initWithWindowNibName:` aufrufen:

ChangingViews/HelloApplication44/MyWindowController.m

```
if ([self shouldLoadIconView])[self loadIconView:self];
else [self loadTableView:self];
```

Speichern Sie Ihre Änderungen und öffnen Sie `MainMenu.xib` im Interface Builder. Verwenden Sie den Connections Inspector, um die Aktionen mit den dazugehörigen Menüpunkten zu verbinden. Speichern Sie ab.

Klicken Sie auf *Build & Run*, und Sie sollten in der Lage sein, über die Menüelemente oder die Tastaturkürzel zwischen den Views hin- und herzuwechseln. Sobald Sie in den Icon-View wechseln, erscheint die Startnotifikation für *HelloApplication*. Das Problem ist die Methode `setUpView`.

ChangingViews/HelloApplication44/MyWindowController.m

```
-(void) setUpView {
►    self.currentApp = [[CurrentApp alloc] init];
     self.currentApp.delegate = self.ac;
     [self.window setContentSize:[self.ac.view bounds].size];
     self.window.contentView = self.ac.view;
►    [self.ac applicationDidLaunch:
►                        [NSRunningApplication currentApplication]];
}
```

Die `setUpView`-Methode war dazu gedacht, nur einmal beim ersten Start der Anwendung aufgerufen zu werden. Nun wird Sie jedesmal aufgerufen, wenn wir den View wechseln. Es gibt keinen Grund dafür, bei jedem Wechsel des Views eine neue Instanz von `CurrentApp` zu erzeugen. Wir wollen auch die Tatsache festhalten, dass die aktuelle Anwendung das erste Mal gestartet wurde.

Wir lösen das erste Problem mithilfe von „lazy initialization“ („fauler Initialisierung“) und das zweite, indem wir herausfinden, ob die aktuelle Anwendung bereits läuft.

18.10 Lazy Initialization

Wir wollen nicht jedesmal eine `CurrentApp`-Instanz erzeugen, wenn wir die Views wechseln. `CurrentApp` soll nur beim Start instanziiert werden, wenn wir es brauchen.

Zwar werden die Getter und Setter für die `currentApp`-Eigenschaft für uns generiert, aber wir können sie natürlich überschreiben. Wir ändern die Getter-Methode `currentApp` so:

ChangingViews/HelloApplication45/MyWindowController.m

```
-(CurrentApp *) currentApp {
    if (!currentApp) {
        self.currentApp = [[CurrentApp alloc] init];
    }
    return currentApp;
}
```

Ist die Instanzvariable `currentApp` nicht `nil`, geben wir den Wert einfach zurück. Wenn sie noch nicht existiert, erzeugen wir eine Instanz von `CurrentApp` und weisen sie unserer Eigenschaft über den Setter zu.

Wir passen die `init`-Methode in `CurrentApp.m` so an, dass ihre `app`-Eigenschaft die aktuelle Anwendung ist:

ChangingViews/HelloApplication45/CurrentApp.m

```
-(id) init {
    if (self =[super init]) {
        [self registerNotifications];
        [self initializeMethodDictionary];
►       self.app = [NSRunningApplication currentApplication];
    }
    return self;
}
```

Nun besitzen wir während der gesamten Laufzeit unserer Anwendung eine einzige Instanz von `CurrentApp`, die immer weiß, was die startende oder endende Anwendung ist.

Es gibt viele Möglichkeiten, um herauszufinden, ob unsere aktuelle Anwendung startet oder endet. Wir könnten `CurrentApp` beispielsweise um eine Eigenschaft erweitern, die diese Information enthält. Wir könnten die Notifikation im View-Controller speichern und anhand des Typs entscheiden. Ich verfolge einen einfacheren Ansatz: Da wir die Anwendung kennen, die gerade gestartet oder beendet wird, überprüfen wir, ob sie in unserem Array laufender Anwendungen steht.

ChangingViews/HelloApplication45/MyWindowController.m

```
-(void)launchOrTerminate {
    if ([[[NSWorkspace sharedWorkspace] runningApplications]
                        containsObject:self.currentApp.app]) {
►       [self.ac applicationDidLaunch:self.currentApp.app];
    } else [self.ac applicationDidTerminate:self.currentApp.app];
}
```

Abschließend – man könnte auch sagen: *endlich* – müssen wir die `setUpView`-Methode überarbeiten. Entfernen Sie die Zeilen am Anfang und am Ende, die bei jedem Aufruf ein neues `CurrentApp`-Objekt erzeugt und `applicationDidLaunch:` mit der momentan laufenden Anwendung als Parameter aufgerufen haben. Fügen Sie außerdem die hervorgehobene Zeile ein, die unsere `launchOrTerminate`:-Methode aufruft:

ChangingViews/HelloApplication45/MyWindowController.m

```
-(void) setUpView {
    self.currentApp.delegate = self.ac;
    [self.window setContentSize:[self.ac.view bounds].size];
►   [self launchOrTerminate];
    self.window.contentView = self.ac.view;
}
```

Klicken Sie auf *Build & Run*. Sie können nun zwischen den Views wechseln, sooft Sie wollen. Die neuesten Informationen werden im Icon-View dargestellt, und die Liste aller laufenden Anwendungen (ohne die gelöschten) im Tabellen-View.

Wir könnten noch so viel mehr anstellen … Wenn das hier eine Produktivanwendung wäre, würde ich die Einstellungen für den View wahrscheinlich entfernen und immer mit dem View starten, den der Benutzer verwendet hat, als er die Anwendung beim letzten Mal verlassen hat. Ich könnte den Icon-View um Animationen für die startenden und endenden Anwendungen erweitern. Die Möglichkeiten für den Feinschliff sind endlos, doch es wird Zeit, sich von diesem Beispiel zu verabschieden.

Wir haben mit diesem Beispiel Vieles abgehandelt. Wir haben auf Notifikationen reagiert und eigene erzeugt. Wir haben unsere eigenen Protokolle und Delegates entwickelt und eingesetzt. Wir haben mit Dictionaries und Tabellen gearbeitet. Wir haben Daten gespeichert und mit Einstellungen gearbeitet. Wir haben unser Nib in kleine Teile zerlegt, einen eigenen View erstellt und zwischen den Nibs hin- und hergewechselt, um unser Einstellungsfenster und unsere Views anzuzeigen.

Das Beispiel hat uns gute Dienste geleistet, aber es wird Zeit, dass wir uns einem neuen Beispiel zuwenden, das uns durch verschiedene Aspekte von Bindungen und Core Data begleiten wird.

Kapitel 19

Key Value Coding

Es ist sehr leicht, den wesentlichen Punkt am Key Value Coding zu vergessen. Die meisten Leute konzentrieren sich auf den Mechanismus und nicht darauf, was er einem eigentlich zu tun erlaubt. Lassen wir das *Wie* ein wenig außen vor, damit wir über das *Warum* und das *Wann* nachdenken können.

Stellen Sie sich eine Klasse namens `PragBook` vor, die eine Eigenschaft namens `title` besitzt. Wäre `jrport` eine Instanz von `PragBook`, würden Sie den Titel so ermitteln:

```
NSString *bookTitle = [jrport title];
```

Sie könnten auch die Punktsyntax nutzen, um auf die Eigenschaft zuzugreifen.

```
NSString *bookTitle = jrport.title;
```

Mit Key Value Coding, das wir von nun an kurz KVC nennen wollen, würden Sie es so formulieren:

```
NSString *bookTitle = [jrport valueForKey:@"title"];
```

Das sieht furchtbar aus. Es gibt viele Aspekte, die ganz offensichtlich schlechter sind als der direkte Ansatz. Erstens müssen Sie mehr schreiben. Zweitens übergeben wir den Namen der Variablen als String, es gibt also keine Prüfung während der Kompilierung. Und drittens stellt sich die Frage, warum jemand auf so seltsame Weise auf Variablen zugreifen sollte.

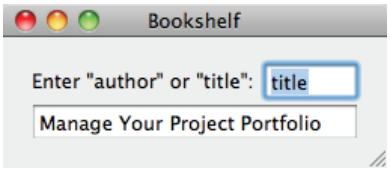
KVC ist der erste Schritt auf unserem Weg zu Key Value Observing, Bindungen und Core Data. In diesem Kapitel werden Sie ein Gefühl dafür entwickeln, was es ist, wie man es benutzt und warum man es überhaupt benutzen will.

19.1 Objekte wie Dictionaries behandeln

Sobald wir uns beruhigt haben, werden wir uns daran erinnern, dass wir eine Notation wie `valueForKey:` schon einmal gesehen haben. Das erinnert stark an die `objectForKey:`-Methode, die wir bei unserer Arbeit mit Dictionaries verwendet haben.

Wir haben angefangen, bestimmte Einträge über fest kodierte Schlüssel aus dem Dictionary abzurufen. Als wir dann bei Kapitel 16, *Daten in einer Tabelle darstellen*, angekommen waren, änderte sich alles schlagartig: In diesem Kapitel haben wir Tabellen gefüllt, indem wir die Spaltenbezeichner als Schlüssel verwendet haben. Die Tabelle konnte so mit dem Inhalt dieser Dictionaries gefüllt werden, ohne viel konditionalen Code schreiben zu müssen. Mit KVC können wir die gleichen Tricks auf Objekte und ihre Variablen anwenden.[1]

Wir werden eine Anwendung entwickeln, die es Ihnen erlaubt, *author* oder *title* in ein Textfeld einzugeben, woraufhin der Name des Autors bzw. der Titel des Buches in einem weiteren Textfeld erscheint. Die GUI sieht so aus:



Legen Sie eine neue Cocoa-Anwendung namens `Bookshelf` an und fügen Sie mit dem folgenden Header eine Objective-C-Klasse namens `PragBook` ein:

KVC/Bookshelf1/PragBook.h

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

@interface PragBook : NSObject {
    NSString *title;
    NSString *author;
}
@property (copy) NSString *title, *author;

@end
```

Da Sie gerade dabei sind, synthetisieren Sie die Akzessoren `title`, `setTitle:`, `author` und `setAuthor:` in der Implementierungsdatei.

1 Sie haben die `valueForKey:`-Methode kurz in Abschnitt 16.8, *Ausblick auf bevorstehende Knüller*, auf Seite 269 gesehen.

KVC/Bookshelf1/PragBook.m

```
#import "PragBook.h"

@implementation PragBook
@synthesize title, author;
@end
```

Damit erhalten wir die Klasse `PragBook` mit den beiden Instanzvariablen `title` und `author` und den dazugehörigen Akzessormethoden. Wir sollten in der Lage sein, diese Variablen irgendwie als Schlüssel zu nutzen. Wir sollten den Wert von `title` auf die gleiche Weise abfragen können, in der wir ein Objekt aus einem Dictionary abgerufen haben.

Genau das bietet uns KVC. Es erlaubt uns den Einsatz von `valueForKey:`, um den Wert zurückzugeben, der zu der als String übergebenen Variable gehört.[2] Wie das funktioniert, ist genial. Nehmen wir an, dass Ihr Code folgenden Aufruf enthält:

```
[jrport valueForKey:@"title"];
```

Das Laufzeitsystem durchsucht die `PragBook`-Klasse nach einer Instanzmethode namens `getTitle`, `title` oder `isTitle` (in dieser Reihenfolge). Die zuerst gefundene wird verwendet. Existieren diese nicht, wird im nächsten Schritt nach Methoden für Kollektionen gesucht – wir überspringen diesen Schritt für's Erste, weil `title` keine Kollektion ist.

Existiert keine dieser einfache Akzessormethoden, wird im nächsten Schritt versucht, direkt den Wert der *Ivar*[3] zu ermitteln. Gibt die `accessInstanceVariablesDirectly`-Methode des Objekts `YES` zurück, dann sucht das Laufzeitsystem nach einer Instanzvariablen namens `_title`, `_isTitle`, `title` oder `isTitle` (in dieser Reihenfolge) und gibt den Wert der Variablen zurück.

Zweierlei sollten Sie noch wissen. Zum einen habe ich einige Details ausgelassen. Ist eine Variable kein Zeiger auf ein Objekt, erfolgt eine Umwandlung in eine `NSNumber` (wenn es denn passt) bzw. in einen `NSValue`. Zum anderen wird die Methode `valueForUndenedKey:` aufgerufen, wenn keine Methode oder Variable gefunden werden konnte.

2 Wir haben für Methoden bereits etwas Vergleichbares gemacht: Wir haben Namen von Methoden, die unter bestimmten Bedingungen aufgerufen werden sollen, in Form von Strings übergeben.

3 Im Englischen wird „ivar", Instanzvariable, wie die Paprikapaste Ajvar ausgesprochen.

Beachten Sie, dass wir durch die Verwendung von Eigenschaften fast keinen Code schreiben mussten, und dennoch die Bedingungen erfüllt haben, die für KVC notwendig sind. Nun wollen wir diese Klasse in einer einfachen Anwendung nutzen.

19.2 Variablen mit KVC abrufen

Fügen Sie eine weitere Objective-C-Klasse namens `BookshelfController` (die `NSViewController` erweitert) ein. Zunächst soll `BookshelfController` eine einzelne `PragBook`-Instanz enthalten. Wir benötigen eine Aktion, die aufgerufen wird, wenn der Benutzer *author* oder *title* eingegeben hat. Wir benötigen außerdem ein Outlet, damit wir den Buchtitel oder Autorennamen in das zweite Textfeld schreiben können. All das findet sich in der Header-Datei für den `BookshelfController`.

KVC/Bookshelf1/BookshelfController.h

```
#import <Cocoa/Cocoa.h>
@class PragBook;

@interface BookshelfController : NSViewController {
    NSTextField *valueField;
    PragBook *book;
}
@property IBOutlet NSTextField *valueField;
@property PragBook *book;

-(IBAction) getValue:(id) sender;
@end
```

Die Implementierung erzeugt eine einzelne Buchinstanz und reagiert auf die Benutzereingaben *author* und *title* im oberen Textfeld, indem sie im unteren Textfeld den Namen des Autors bzw. den Titel des Buches ausgibt.

KVC/Bookshelf1/BookshelfController.m

```
#import "BookshelfController.h"
#import "PragBook.h"

@implementation BookshelfController
@synthesize book, valueField;

-(IBAction) getValue:(id) sender {
    [self.valueField setStringValue:
                        [self.book valueForKey:[sender stringValue]]];
}

-(PragBook *) book {
    if (!book) {
        self.book = [[PragBook alloc] init];
```

```
        self.book.title = @"Manage Your Project Portfolio";
        self.book.author = @"Johanna Rothman";
    }
    return book;
}
@end
```

Ich brauche noch eine Minute, um zu beschreiben, welche Arbeiten noch notwendig sind, um diese Anwendung ans Laufen zu bringen, und dann widme ich mich mit großem Tamtam der `getValue:`-Aktion.

Legen Sie eine neue, View-basierte *.nib*-Datei an und nennen Sie sie `Bookshelf`. Bauen Sie die GUI auf, indem Sie ein Label und die beiden Textfelder wie in der obigen Abbildung anordnen. Nutzen Sie den Attributes Inspector, um die Selektierung des zweiten Textfeldes zu unterbinden und die Aktion des ersten Textfeldes nur beim Drücken der *Enter*-Taste anzustoßen.

Verwenden sie den Identity Inspector, um als Typ des *File's Owner* `Bookshelf-Controller` anzugeben. Als Nächstes verwenden Sie den Connections Inspector, um das `valueField`-Outlet mit dem unteren und die `getValue:`-Aktion mit dem oberen Textfeld zu verknüpfen. Verbinden Sie das `view`-Outlet mit ihrem View. Speichern Sie Ihre Änderungen.

Erzeugen Sie eine Instanz ihres View-Controllers und laden Sie das `Bookshelf`-Nib in der `applicationDidFinishLaunching:`-Methode ihres App-Delegate. Passen Sie die Größe des Fensters an und legen Sie als seinen Content-View den View des View-Controllers fest.

KVC/Bookshelf1/BookshelfAppDelegate.m

```
#import "BookshelfAppDelegate.h"
#import "BookshelfController.h"

@implementation BookshelfAppDelegate

@synthesize window;

-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
   BookshelfController * bc = [[BookshelfController alloc]
                               initWithNibName:@"Bookshelf" bundle:nil];

    [self.window setContentSize:[bc.view bounds].size];
    self.window.contentView = bc.view;
}

@end
```

Klicken Sie auf *Build & Run*. Geben Sie *title* in das obere Textfeld ein, und sobald Sie *F* drücken, sollte „Manage Your Project Portfolio" im unteren Textfeld erscheinen. Geben Sie stattdessen *author* ein, erscheint „Johanna Rothman". Wir haben KVC benutzt, um den Wert einer Variablen abzurufen. Ich liebe die Einfachheit und Flexibilität dieses Aufrufs der `getValue:`-Methode.

KVC/Bookshelf1/BookshelfController.m

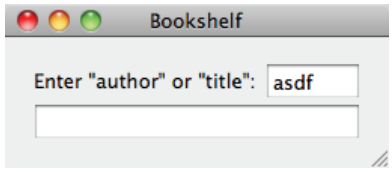
```
[self.valueField setStringValue:
                  [self.book valueForKey:[sender stringValue]]];
```

Sehen wir uns das mal genauer an. Die `getValue:`-Methode wird aufgerufen, wenn der Benutzer die *Enter*-Taste drückt, nachdem er *author* oder *title* eingegeben hat. Das erste Textfeld übergibt einen Zeiger auf sich selbst als `sender`-Argument, was wir nutzen, um den vom Benutzer eingegebenen String zu ermitteln. Als Nächstes verwenden wir diesen String als Schlüssel und suchen nach dem Wert einer Eigenschaft mit diesem Namen in `book`. Zum Schluss übergeben wir diesen Wert an das zweite Textfeld, um ihn als String ausgeben zu lassen.

Nehmen wir einmal an, Sie erweitern `PragBook` um eine Eigenschaft für den Untertitel und vielleicht noch eine für Kommentare wie: „Ich habe gelacht und geweint; es hat mich sehr bewegt." Sie müssen keine Änderung an der *.nib*-Datei und auch nicht an der `getValue:`-Methode vornehmen. Sie müssen nur die `PragBook`-Klasse anpassen.

19.3 Undefinierte Schlüssel

Unsere Anwendung hat ein grundsätzliches Problem. Zwar soll der Benutzer *author* oder *title* in das erste Textfeld eingeben, doch er kann eingeben, was er will. Probieren Sie es, geben Sie *asdf* in das erste Textfeld ein.



Der Benutzer erhält keinerlei Feedback. Er weiß nicht, dass er etwas falschgemacht hat. Wir als Entwickler sehen andererseits Folgendes in der Debugger-Konsole:

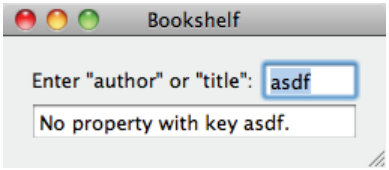
```
Exception detected while handling key input.
[<PragBook 0x1001068f0> valueForUndefinedKey:]:
        this class is not key value coding-compliant for the key asdf.
```

Um das zu beheben, muss unser `PragBook` die Methode `valueForUndefinedKey:` implementieren. In diesem Fall geben wir eine Fehlermeldung als String zurück. Bei Bedarf können Sie (basierend auf dem vom Benutzer eingegebenen Schlüssel) auch andere Objekte zurückgeben. Hier geben wir für alle Fehler den gleichen String zurück.

KVC/Bookshelf2/PragBook.m

```
#import "PragBook.h"

@implementation PragBook
@synthesize title, author;

► -(id) valueForUndefinedKey:(NSString *)key {
►    return [NSString stringWithFormat:@"No property with key %@.", key];
► }
@end
```

Mit dieser einfachen Änderung geben wir den Fehler im zweiten Textfeld aus.



Auch wenn es ganz nett ist, in dieser Form mit unerwünschten Benutzereingaben umgehen zu können, würden wir die Schnittstelle doch besser so entwerfen, dass der Benutzer keine undefinierten Schlüssel eingeben kann. Sie könnten das mit einer vordefinierten Drop-down-Liste erledigen, oder Sie könnten eine Schnittstelle mit festen Feldern verwenden – eines für jede darzustellende Eigenschaft.

19.4 Übung: Variablen setzen per KVC

Es gibt eine Setter-Methode namens `setValue:forKey:`, und die zugehörige Getter-Methode `valueForKey:`. Erweitern Sie den `BookshelfController` um eine Aktion namens `setValue:`.

In dieser Methode setzen Sie den Wert der im oberen Textfeld eingegebenen Eigenschaft auf den im unteren Textfeld eingegebenen Wert. Mit anderen Worten gibt der Benutzer ein Schlüssel/Wert-Paar ein, dessen Schlüssel (Autor oder Titel) im oberen Textfeld steht, während der diesem Schlüssel zugewiesene Wert im unteren Textfeld steht.

19.5 Lösung: Variablen setzen per KVC

Zugegeben, das ist ein albernes Beispiel, aber es verdeutlicht die Abfrage und das Setzen von Eigenschaften mit KVC. Sie müssen ein Outlet und eine Aktion in die `BookshelfController`-Header-Datei einfügen:

KVC/Bookshelf3/BookshelfController.h

```
#import <Cocoa/Cocoa.h>
@class PragBook;

@interface BookshelfController : NSViewController {
►    NSTextField *keyField;
    NSTextField *valueField;
    PragBook *book;
}
► @property IBOutlet NSTextField *valueField, *keyField;
@property PragBook *book;

-(IBAction) getValue:(id) sender;
► -(IBAction) setValue:(id) sender;
@end
```

Verbinden Sie das neue Outlet und die Aktion in der *.nib*-Datei. Legen Sie das untere Textfeld als selektier- und editierbar fest. Außerdem soll es seinen Wert senden, sobald die *Enter*-Taste gedrückt wird.

Synthetisieren Sie die `keyField`-Eigenschaft in der Implementierungsdatei und implementieren Sie die `setValue:`-Aktion so:

KVC/Bookshelf3/BookshelfController.m

```
-(IBAction) setValue:(id) sender {
    [self.book setValue:[sender stringValue]
                 forKey:[self.keyField stringValue]];
}
```

So können Sie den Wert jeder definierten Eigenschaft abrufen und setzen, als würde es sich um Schlüssel und Werte eines Dictionary handeln. Wir wollen jetzt noch einen Schritt weiter gehen.

19.6 KVC und Dictionaries

Sie haben also gesehen, dass das Key Value Coding es Ihnen erlaubt, Eigenschaften und deren Werte so zu behandeln, als wären sie Einträge in einem Dictionary. Wir wollen jetzt einen Schritt weiter gehen. Wie sich zeigt, ist es ganz einfach, ein Dictionary zu nutzen, um die Eigenschaften eines Objekts festzulegen bzw. ein Dictionary aus einer Klasse zu erzeugen, solange die Schlüssel den Namen der Eigenschaften entsprechen.

Wir wollen unser Bookshelf-Beispiel so abändern, dass das PragBook instanziiert und mit Werten aus einem Dictionary befüllt wird. Statt die Eigenschaften in book zu initialisieren, erzeugen wir diesmal ein Dictionary namens bookInfo mit den Schlüsseln title und author. Nun setzen wir die Werte der book-Eigenschaften alle auf einmal über das bookInfo-Dictionary:

KVC/Bookshelf4/BookshelfController.m

```
-(PragBook *) book {
    if (!book) {
        self.book = [[PragBook alloc] init];
        NSDictionary *bookInfo =
                        [NSDictionary dictionaryWithObjectsAndKeys:
                        @"Manage Your Project Portfolio",@"title",
                        @"Johanna Rothman",@"author", nil];
        [self.book setValuesForKeysWithDictionary: bookInfo];
    }
    return book;
}
```

Das war's.

Der umgekehrte Weg ist etwas aufwendiger. Wir müssen ein Array übergeben, dass die Schlüssel des neuen Dictionary enthält. Diese entsprechen den Objekteigenschaften, die wir in diesem Dictionary festhalten wollen. In unserem Fall wollen wir beide festhalten. Fügen Sie die folgende Zeile in setValue: ein, um das Dictionary zu erzeugen und zu protokollieren, wenn der Benutzer den Wert von title oder author ändert:

KVC/Bookshelf5/BookshelfController.m

```
-(IBAction) setValue:(id) sender {
    [self.book setValue:[sender stringValue]
                 forKey:[self.keyField stringValue]];
    NSLog(@"%@",[self.book dictionaryWithValuesForKeys:
                  [NSArray arrayWithObjects:@"author",@"title", nil]]);
}
```

Hier habe ich den Wert der author-Variablen von Johanna Rothman in JR geändert:

```
Bookshelf[6388:a0f] {
    author = JR;
    title = "Manage Your Project Portfolio";
}
```

Auch dieses Beispiel kommt einem in dieser einfachen Form albern vor, doch man kann sich leicht nützliche Anwendungsfälle vorstellen. Nehmen wir zum Beispiel an, dass Sie die Buchinformationen auf der Fest-

platte speichern wollen, sobald ein Wert verändert wurde. Sie wissen bereits, wie man Dictionaries auf die Festplatte schreibt und wieder einliest, und jetzt wissen Sie auch, wie man eine KVC-konforme Klasse in ein Dictionary schreibt und wieder ausliest. Das ermöglicht Ihnen das Speichern und Abrufen jeder KVC-konformen Klasse.

19.7 Schlüsselpfade für die Navigation in einer Klassenhierarchie

Im Moment enthält unser Bücherregal nur ein Buch. Nun wollen wir unser Buch um ein Kapitel erweitern. Das `Chapter`-Objekt wird Variablen für den Titel und die Anzahl der Seiten besitzen. Ich möchte Ihnen zeigen, wie man eine Hierarchie mit KVC durchgeht, indem man Schlüsselpfade (keypaths) anstelle von Schlüsseln benutzt.

Legen Sie eine neue Objective-C-Klass namens `Chapter` an. Hier ist die Header-Datei:

KVC/Bookshelf6/Chapter.h

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

@interface Chapter : NSObject {
    NSString *chapterTitle;
    NSNumber *pageCount;
}
@property(copy) NSString *chapterTitle;
@property(copy) NSNumber *pageCount;
@end
```

Erneut machen wir in der Implementierungsdatei nicht viel mehr, als die Eigenschaften zu synthetisieren:

KVC/Bookshelf6/Chapter.m

```
#import "Chapter.h"

@implementation Chapter
@synthesize chapterTitle, pageCount;

@end
```

Jetzt bewegen wir uns eine Ebene höher und fügen eine Instanz von `Chapter` in `PragBook` ein:

KVC/Bookshelf6/PragBook.h

```
#import <Cocoa/Cocoa.h>
@class Chapter;
```

```
@interface PragBook : NSObject {
    NSString *title;
    NSString *author;
►   Chapter *chapter;
}
@property (copy) NSString *title, *author;
► @property Chapter *chapter;
@end
```

Synthetisieren Sie die Akzessoren für `chapter`. Erzeugen und initialisieren Sie `chapter` in der `init`-Methode von `PragBook`:

KVC/Bookshelf6/PragBook.m

```
#import "PragBook.h"
► #import "Chapter.h"

@implementation PragBook
► @synthesize title, author, chapter;

► -(id)init {
►     if (self=[super init]) {
►         self.chapter = [[Chapter alloc] init];
►     }
►     return self;
► }
-(id) valueForUndefinedKey:(NSString *)key {
   return [NSString stringWithFormat:@"No property with key %@.", key];
}
@end
```

Unser `BookshelfController`-Objekt besitzt nun eine einzelne `PragBook`-Instanz, die wiederum über eine einzelne `Chapter`-Instanz mit den Eigenschaften `chapterTitle` und `pageCount` verfügt. Wir wollen KVC verwenden, um die Werte des Titels und der Seitenzahl in der `book`-Methode in `BookshelfController.m` zu setzen. Ich habe in den hervorgehobenen Zeilen `setValue:forKeyPath:` zweimal genutzt.

KVC/Bookshelf6/BookshelfController.m

```
-(PragBook *) book {
    if (!book) {
        self.book = [[PragBook alloc] init];
        NSDictionary *bookInfo =
                        [NSDictionary dictionaryWithObjectsAndKeys:
                        @"Manage Your Project Portfolio",@"title",
                        @"Johanna Rothman",@"author", nil];
        [self.book setValuesForKeysWithDictionary: bookInfo];
►       [self.book setValue:@"Preface"
►                 forKeyPath:@"chapter.chapterTitle"];
►       [self setValue:[NSNumber numberWithInt:4]
►           forKeyPath:@"book.chapter.pageCount"];
    }
    return book;
}
```

Ich habe zwei verschiedene Pfade verwendet, um die Sache zu veranschaulichen. Wenn ich mit dem Buch beginne, lautet der Schlüsselpfad `chapter.chapterTitle`. Beginne ich hingegen bei *self* (also auf `BookshelfController`-Ebene), ist `book.chapter.pageCount` der verwendete Pfad.

Beachten Sie, dass wir die Methode `setValue:forKeyPath:` anstelle von `setValue:forKey:` verwenden. Tatsächlich können Sie die Schlüssel als sehr kurze Schlüsselpfade betrachten und `setValue:forKeyPath:` in unserer `setValue:`-Aktion verwenden.

KVC/Bookshelf6/BookshelfController.m

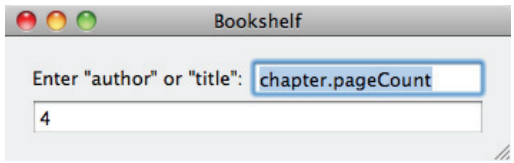
```
-(IBAction) setValue:(id) sender {
►     [self.book setValue:[sender stringValue]
►               forKeyPath:[self.keyField stringValue]];
}
```

In der gleichen Weise wird der Getter `valueForKeyPath:` anstelle von `valueForKey:` verwendet. Das erlaubt uns die Angabe von Schlüsseln oder Schlüsselpfaden, wo wir vorher nur Schlüssel angeben konnten.

KVC/Bookshelf6/BookshelfController.m

```
-(IBAction) getValue:(id) sender {
►     [self.valueField setStringValue:
►                       [self.book valueForKeyPath:[sender stringValue]]];
}
```

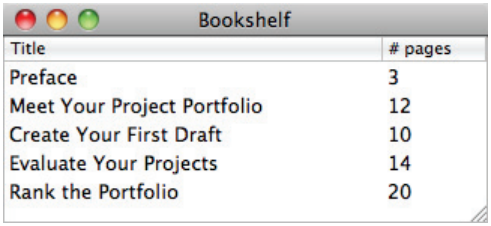
Neben der Eingabe der Autoren- und Titelschlüssel im oberen Textfeld können wir nun auch die Schlüsselpfade „chapter.chapterTitle" und „chapter.pageCount" verwenden, um auch diese Werte zu setzen und abzurufen.



Schlüsselpfade sind viel weniger mysteriös, jetzt wo wir Eigenschaften und die Punktnotation kennen. Das Key Value Coding funktioniert sogar ohne Eigenschaften, solange Ihre Klasse für diesen Schlüssel Key Value Coding-konform ist. Die Apple-Dokumentation erläutert, dass im Fall eines Attributs namens „key" ein Getter namens `key` oder `isKey` bzw. eine Ivar namens `key` oder `_key`. existieren muss. Wenn Sie einen Setter brauchen, muss sein Name `setKey:` lauten. Die entsprechende Methode führt keine Validierung durch. Bisher haben wir die Eingabe nicht validiert, doch wenn Sie es brauchen, ist das die Aufgabe der Methode `validateKey:error:`.

19.8 Übung: Tabellen füllen mit KVC

KVC erlaubt uns, eine Tabelle ebenso einfach aufzufüllen, wie wir das in einem früheren Beispiel mit einem Dictionary getan haben.



Füllen Sie die Tabelle über ein Kapitel-Array auf. Dieses Array fügen Sie in die `Prag-Book`-Klasse anstelle der bisher vorhandenen `chapter`-Eigenschaft ein. Der `BookshelfController` wird Datenquelle und Delegate für den Tabellen-View sein. Welche Methoden müssen Sie implementieren? Wie müssen diese implementiert werden?

19.9 Lösung: Tabellen füllen mit KVC

Wir wollen mit dem `Bookshelf`-Nib beginnen. Ziehen Sie einen Tabellen-View in Ihr Dokumentenfenster. Sie können Ihren eigenen View zusammen mit seinem Label und den Textfeldern löschen. Legen Sie im Attributes Inspector `chapterTitle` als Bezeichner der ersten Spalte und `pageCount` als den der zweiten fest. Legen Sie `BookshelfController` als Delegate und die Datenquelle des Tabellen-Views fest und verbinden Sie das View-Outlet des `BookshelfController` mit dem Scroll-View. Speichern Sie ab.

Wenn Sie *Build & Run* anklicken, müsste die Anwendung starten, und das Fenster mit dem Tabellen-View müsste erscheinen. Doch in der Konsole sollte der folgende Fehler zu sehen sein:

```
Illegal NSTableView data source (<BookshelfController: 0x200092c60>).
Must implement numberOfRowsInTableView: and
               tableView:objectValueForTableColumn:row:
```

Wenn wir davon ausgehen, dass `PragBook` eine Eigenschaft namens `chapters` besitzt, bei der es sich um ein `NSArray` voller `Chapter`-Objekte handelt, dann implementieren wir `numberOfRowsInTable-View:`, indem wir die Anzahl der Elemente im `chapters`-Array zurückliefern. An dieser Stelle KVC zu verwenden, bringt uns keinen Vorteil:

KVC/Bookshelf7/BookshelfController.m

```
-(NSInteger) numberOfRowsInTableView:(NSTableView *) aTableView {
    return [self.book.chapters count];
}
```

Wir benutzen KVC, um jede Zelle des Tabellen-Views zu füllen:

KVC/Bookshelf7/BookshelfController.m

```
-(id)tableView:(NSTableView *)aTableView
                objectValueForTableColumn:(NSTableColumn *)aTableColumn
                                      row:(NSInteger)rowIndex {
    return [[self.book.chapters objectAtIndex:rowIndex]
►                               valueForKey:[aTableColumn identifier]];
}
```

Der Vollständigkeit halber fügen Sie die folgenden (der Bequemlichkeit dienenden) Methoden zur Erzeugung neuer Objekte in `Chapter` ein:

KVC/Bookshelf7/Chapter.m

```
-(id) initWithTitle:(NSString *) title pageCount: (int) count {
    if (self=[super init]) {
        self.chapterTitle = title;
        self.pageCount = [NSNumber numberWithInt:count];
    }
    return self;
}
+(id) chapterWithTitle:(NSString *) title pageCount: (int)count {
    return [[Chapter alloc ] initWithTitle:title
                                 pageCount:count];
}
```

Fügen Sie ein Array namens `chapters` zu `PragBook` hinzu. Sie können die `chapter`-Variable vom Typ `Chapter` löschen. Füllen Sie das `chapters`-Array in der `init`-Methode von `PragBook` mit einigen Kapiteln auf.[4]

KVC/Bookshelf7/PragBook.m

```
-(id)init {
    if (self=[super init]) {
        self.chapters = [[NSArray alloc] initWithObjects:
            [Chapter chapterWithTitle:@"Preface"
                            pageCount:3],
            [Chapter chapterWithTitle:@"Meet Your Project Portfolio"
                            pageCount:12],
            [Chapter chapterWithTitle:@"Create Your First Draft"
                            pageCount:10],
            [Chapter chapterWithTitle:@"Evaluate Your Projects"
                            pageCount:14],
```

4 Entfernen Sie den Code zur Einrichtung einer einzelnen Kapitelüberschrift und der Seitenzahl aus der `book`-Methode.

```
            [Chapter chapterWithTitle:@"Rank the Portfolio"
                                pageCount:20], nil];
    }
    return self;
}
```

Das sollte an Informationen reichen, um das Projekt ans Laufen zu bringen. Sie müssen die passenden Deklarationen in die Header-Dateien eintragen und einige der Methoden implementieren. Sie müssen außerdem die Outlets und Aktionen aus dem `BookshelfController` entfernen.[5] Wenn Sie hängenbleiben, können Sie jederzeit im Codedownload unter `KVC/Bookshelf7` nachsehen.

19.10 Arrays und KVC

Im vorigen Abschnitt haben wir uns bestimmte Elemente in unserem Array so herausgepickt:

```
[[self.book.chapters objectAtIndex:rowIndex]
                            valueForKey:[aTableColumn identifier]];
```

Der erste Teil wählt ein bestimmtes Kapitel und der zweite pickt sich eine der Eigenschaften heraus. Wir können noch mehr tun. Zum Beispiel können wir mit einem Aufruf ein Array erzeugen, das alle Kapitelüberschriften enthält:

```
[self.book.chapters valueForKey:@"chapterTitle"]
```

Da ist mehr dran, als man auf den ersten Blick sieht. Wir besitzen ein Array namens `chapters`, dessen Einträge alle Objekte vom Typ `Chapter` sind. Jedes Kapitel besitzt zwei Eigenschaften: `chapterTitle` und `pageCount`. Stellen Sie sich die Schritte vor, die Sie in der Vergangenheit hätten unternehmen müssen, um dieses Array zu füllen: Sie hätten das neue Array erzeugen und die Elemente aus dem alten Array kopieren müssen. Das ist keine große Sache, aber KVC macht sie klarer und einfacher.

Doch das ist nicht alles.

Sie können mit diesen neuen Kollektionen arbeiten und alle Arten von Berechnungen durchführen. Die Eigenschaft `pageCount` ist beispielsweise eine Zahl, Sie könnten also die Summe aller Seiten oder den Durchschnitt berechnen.

---

5 Machen Sie nicht den (naheliegenden) Fehler, `book.chapter` statt `book.chapters` zu verwenden. Sie sollten die Variable namens `chapter` entfernt haben und über eine Suche leicht sicherstellen können, dass Sie nur die Plural-Variante `chapters` verwenden.

Die Summe würden Sie so berechnen:

```
[self.book valueForKeyPath:@"chapters.@sum.pageCount"]
```

Ist das nicht cool? Sie fügen im Schlüsselpfad einfach `@sum` vor `pageCount` ein. Im Allgemeinen fügen Sie den Operator dem Schlüsselpfad des Arrays hinzu und den Schlüsselpfad der Eigenschaft.

Es gibt zwei Arten von Operatoren. Der erste Typ arbeitet mit Kollektionen von Zahlen und gibt Durchschnitt, Maximum, Minumum und Summe zurück (`@avg`, `@max`, `@min` und `@sum`). Der zweite arbeitet mit Kollektionen und gibt `@count`, `@distinctUnionOfArrays`, `@distinctUnionOfObjects`, `@distinctUnionOfSets`, `@unionOfArrays`, `@unionOfObjects` und `@unionOfSets` zurück.

Wir können beispielhaft einige dieser Operatoren in unserer Anwendung benutzen. Deklarieren Sie eine Instanzmethode namens `createReport` in der `BookshelfController`-Header-Datei und implementieren Sie sie wie folgt:

KVC/Bookshelf8/BookshelfController.m

```
-(void)createReport {
    NSLog(@"There are %@ chapters.",
          [self valueForKeyPath:@"book.chapters.@count.chapterTitle"]);
    NSLog(@"The titles are: %@",
          [self.book.chapters valueForKey:@"chapterTitle"]);
    NSLog(@"This book has %@ pages so far.",
          [self.book valueForKeyPath:@"chapters.@sum.pageCount"]);
    NSLog(@"The longest chapter is %@ pages long.",
          [self valueForKeyPath:@"book.chapters.@max.pageCount"]);
    NSLog(@"The average chapter length is %@.",
          [self.book.chapters valueForKeyPath:@"@avg.pageCount"]);
}
```

Sie werden bemerkt haben, dass ich erneut andere Anker für die Schlüsselpfade verwendet habe, um Ihnen zu zeigen, wo der Operator einzufügen ist. Rufen Sie diese Methode in ihrem App-Delegate gegen Ende der `applicationDidFinishLaunching:`-Methode auf.

KVC/Bookshelf8/BookshelfAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    BookshelfController * bc = [[BookshelfController alloc]
                                initWithNibName:@"Bookshelf" bundle:nil];
    [self.window setContentSize:[bc.view bounds].size];
    self.window.contentView = bc.view;
►    [bc createReport];
}
```

Klicken Sie *Build & Run* an, und Ihre Konsole zeigt so etwas wie das hier:

```
There are 5 chapters.
The titles are: (
    "Preface",
    "Meet Your Project Portfolio",
    "Create Your First Draft",
    "Evaluate Your Projects",
    "Rank the Portfolio"
)
This book has 59 pages so far.
The longest chapter is 20 pages long.
The average chapter length is 11.8.
```

Und, habe ich Sie überzeugt? Erkennen Sie, dass es bei KVC um weit mehr geht, als `[foo bar]` durch `[foo valueForKey:@"bar"]` zu ersetzen? Mehr über KVC erfahren Sie in Apples *Key-Value Coding Programming Guide* [App08g].

KVC bringt sehr viel Flexibilität, doch das ist nur die halbe Wahrheit. Die eigentliche Stärke von KVC sind die anderen Techniken, die dadurch möglich werden. Im nächsten Kapitel sehen wir uns das andere grundlegende Teil dieses Puzzles an: Key Value Observing.

Kapitel 20

Key Value Observing

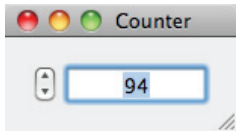
Als Kind habe ich immer auf Schnee gehofft. Manchmal hat es so stark geschneit, dass die Schule geschlossen blieb und wir den Tag damit verbringen konnten, draußen im Schnee zu spielen. Wenn es schneite, hörten wir damals Radio oder sahen fern und warteten auf die Liste der geschlossenen Schulen. Wir mussten unsere Schule dann aus der Liste heraussuchen – der Radiosprecher und der Wettermann wussten ja nicht, welche Schule uns interessierte.

Bei meiner Tochter läuft das heute etwas anders. Wir können die Schulschließungen immer noch im Radio oder Fernsehen verfolgen, aber wir können uns beim lokalen Fernsehsender auch auf der Website registrieren. Jedes Jahr besuche ich die Site und trage meine E-Mail-Adresse und die Schule ein, die für mich relevant ist.

Ich registriere mich als Beobachter (engl. observer). Meine E-Mail-Adresse steht auf einer Liste, die benachrichtigt wird, wenn sich der Wert der `Cocoa Valley-Schulen` von `"offen"` auf einen anderen Wert ändert. In diesem Kapitel sehen wir uns Key Value Observing („Schlüsselwertbeobachtung“, KVO) an, das für Cocoa-Anwendungen die Version des Observer-Musters darstellt.

20.1 Codefreie Verbindungen

Wir wollen dieses Kapitel mit einer einfachen Anwendung beginnen.

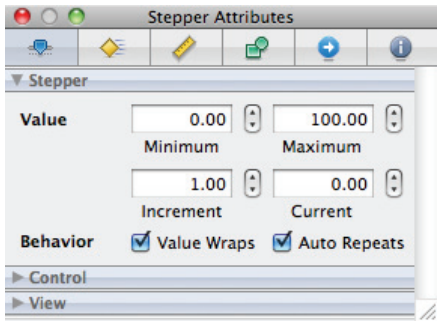


Links sehen Sie einen Schrittzähler (Stepper) und rechts ein Textfeld. Klicken Sie den oberen Teil des Zählers an, erhöht sich die Zahl im Textfeld um eins, und wenn Sie den unteren Teil anklicken, wird der Wert im Textfeld um eins verringert.

Wie würden Sie das programmieren? Tatsächlich wollen wir zuerst ganz ohne Code erledigen.

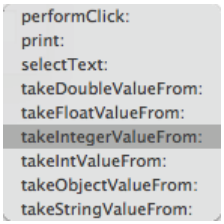
Legen Sie ein neues Cocoa-Projekt namens *Counter* an. Wir wollen für dieses Beispiel ein einzelnes Nib verwenden, also klicken Sie `MainMenu.xib` doppelt an, ziehen Sie einen `NSStepper` und ein Textfeld in das Fenster und passen Sie die Größe so an, dass es wie auf dem Bild aussieht. Der `NSStepper` enthält den Wert, den Sie im Textfeld darstellen wollen. Sie müssen nur die Grenzen festlegen und die Verbindung herstellen.

Als Grenzwerte des Steppers wollen wir einfach die Standardwerte benutzen. Sie können den Stepper anklicken und im Attributes Inspector beobachten, das der Stepper-Wert bei 0 beginnt und in Einserschritten bis 100 hochgeht. Aktivieren Sie die *Behavior*-Checkbox, damit der Wert an seinen Grenzen umspringt.



Nun wollen wir die Verbindung herstellen. Wählen Sie den Stepper. Im Connections Inspector wählen Sie in *Sent Actions* mit Control-Klick den Kreis rechts neben `selector`. Ziehen Sie die Verbindung in das Textfeld und lassen Sie die Maus los.

Sie sollten die folgenden Optionen sehen:

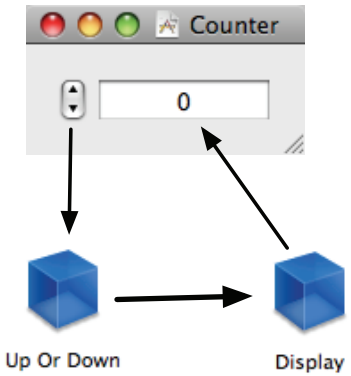


Wählen Sie `takeIntegerValueFrom:`. Im letzten Schritt klicken Sie das Textfeld an und verwenden den Attributes Inspector, um seinen Wert auf `0` zu setzen. Auf diese Weise erscheint der Anfangswert, wenn der Benutzer die Anwendung ausführt. Speichern Sie ab und kehren Sie zu Xcode zurück. Klicken Sie auf *Build & Run*. Sie sollten einen funktionierenden Zähler ohne eine einzige Zeile Programmcode besitzen.

Bislang findet keine Beobachtung statt. Wenn der Benutzer den Stepper anklickt, sendet dieser einfach eine Nachricht an das Textfeld. Das ist der traditionelle Target/Action-Ansatz. Im nächsten Abschnitt machen wir das mit Programmcode noch deutlicher, damit Sie sehen können, in welcher Richtung die Nachricht gesendet wird. Dann werden wir das Modell auf den Kopf stellen und einen Observer einführen.

20.2 Ein Target/Action-Zähler

Bevor wir einen Observer einrichten, wollen wir das Beispiel um zwei Objekte zwischen Stepper und Textfeld erweitern, damit deutlich wird, dass wir momentan mit Target/Action arbeiten. Der Stepper wird eine Nachricht an ein `UpOrDown`-Objekt senden, das eine Nachricht an ein `Display`-Objekt schickt, das wiederum eine Nachricht an das Textfeld sendet. Der Fluss sieht in etwa so aus:



Das sieht vielleicht nach einem Schritt in die falsche Richtung aus, aber ich glaube, dass Sie dann besser erkennen können, was ihnen KVO bietet.

Legen Sie eine neue Objective-C-Klasse namens `Display` an. Diese Klasse benötigt ein Outlet, das Sie mit dem Textfeld verbinden, sowie eine Methode, die vom `UpOrDown`-Objekt aufgerufen wird, um einen neuen Wert für das Textfeld zu übergeben.

Beachten Sie, dass es keine Eigenschaft für das `displayField`, gibt: Wir legen die Variable also als `IBOutlet` fest. Gäbe es eine entsprechende Eigenschaft, könnte man technisch das `IBOutlet`-Label bei der Deklaration der Instanzvariablen beibehalten, aber es ist besserer Stil, das auf die Deklaration der Eigenschaft zu verlagern.

KVO/Counter2/Display.h

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

@interface Display : NSObject {
    IBOutlet NSTextField *displayField;
}
-(void)updateDisplay: (NSNumber *) newValue;
@end
```

Die Implementierung von `Display` enthält nur die Methode, die das Textfeld aktualisiert:

KVO/Counter2/Display.m

```
#import "Display.h"

@implementation Display

-(void)updateDisplay: (NSNumber *) value{
    [displayField setIntegerValue:[value integerValue]];
}

@end
```

Legen Sie außerdem eine Objective-C-Klasse namens `UpOrDown` an, die die Aktion vom Stepper entgegennimmt und an das `Display`-Objekt weitergibt. `UpOrDown` benötigt ein Outlet für die Verbindung mit dem `Display` und eine Aktionsmethode:

KVO/Counter2/UpOrDown.h

```
#import <Cocoa/Cocoa.h>
@class Display;

@interface UpOrDown : NSObject {
    IBOutlet Display *display;
}
-(IBAction) step:(id) sender;
@end
```

Implementieren Sie die Aktion für den Aufruf der `Display`-Methode `updateDisplay:`:

KVO/Counter2/UpOrDown.m

```
#import "UpOrDown.h"
#import "Display.h"

@implementation UpOrDown

-(IBAction) step: (id) sender {
    [display updateDisplay:
        [NSNumber numberWithInteger:[sender integerValue]]];
}
@end
```

Sichern Sie Ihre Arbeit, wechseln Sie zum Interface Builder und verbinden Sie alle miteinander.

Fügen Sie ein Objekt vom Typ `Display` und ein Objekt vom Typ `UpOrDown` in Ihr Dokumentenfenster ein. Im Connections Inspector heben Sie zuerst die direkte Verbindung zwischen Stepper und Textfeld auf. Klicken Sie `UpOrDown` an und verbinden Sie das `display`-Outlet mit dem `Display`-Objekt. Verbinden Sie die `step:`-Aktion mit dem Stepper. Klicken Sie das `Display` an. Verbinden Sie das `displayField`-Outlet mit dem Textfeld. Speichern Sie Ihre Änderungen und klicken Sie auf *Build & Run*.

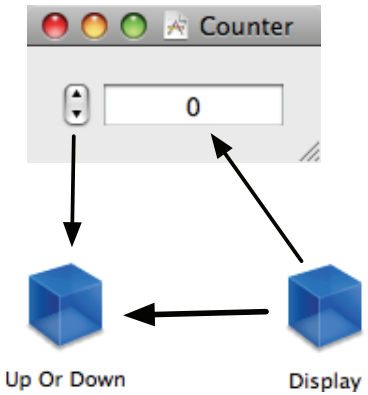
Für den Benutzer ist kein Unterschied zwischen dieser und der code-freien Version zu erkennen.

Beachten Sie, dass das `Display`-Objekt nichts über das `UpOrDown`-Objekt wissen muss. Andererseits muss das `UpOrDown`-Objekt allerdings etwas über das `Display`-Objekt wissen. Als Nächstes wollen wir das auf den Kopf stellen und aus dem `Display`-Objekt einen Observer machen.

20.3 Einen Observer einführen

Beim traditionellen Target/Action-Ansatz besitzt das `UpOrDown`-Objekt ein Handle auf das `Display`-Objekt und sendet die Nachricht `updateDisplay:` an dieses Objekt. Mit KVO weiß der Observer, wen er beobachtet. Das beobachtete Objekt sendet einfach eine Nachricht, wenn sich sein Zustand ändert, und alle registrierten Observer erhalten diese Notifikation.

In unserem Beispiel muss das `Display`-Objekt eine Nachricht an das `UpOrDown`-Objekt senden, um sich als Observer zu registrieren. Das bedeutet, dass unser `Display`-Objekt ein Handle auf das `UpOrDown`-Objekt benötigt. An diesem Punkt sieht die Beziehung wie folgt aus:



Mit anderen Worten haben wir die Beziehung zwischen `UpOr-Down` und dem Stepper sowie zwischen dem `Display` und dem Textfeld nicht geändert. Doch nun muss `Display` über `UpOrDown` Bescheid wissen, während es vorher genau umgekehrt war.

Unser Diagramm zeigt ein Problem auf, das wir vorher nicht hatten: Nichts und niemand hält am `Display`-Objekt fest. Unsere Anwendung läuft mit aktiviertem Garbage Collector. Das `Display`-Objekt registriert sich selbst als Observer und wird dann vom Garbage Collector aussortiert, wenn wir nicht an diesem Objekt festhalten.

Fügen Sie eine Eigenschaft in den App-Delegate ein:

KVO/Counter3/CounterAppDelegate.h

```
#import <Cocoa/Cocoa.h>
► @class Display;

@interface CounterAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
►   Display *display;
}
```

```
@property (assign) IBOutlet NSWindow *window;
@property IBOutlet Display *display;
@end
```

Synthetisieren Sie die Eigenschaft in der Implementierungsdatei und verbinden Sie das Outlet im Interface Builder mit dem `Display`-Objekt.

Als Nächstes verbinden wir das `Display`-Objekt mit dem `UpOrDown`-Objekt und bereiten die Objekte für den Observer vor. Zuerst fügen wir ein `IBOutlet` in den `Display`-Header ein:

KVO/Counter3/Display.h

```
#import <Cocoa/Cocoa.h>
@class UpOrDown;

@interface Display : NSObject {
    IBOutlet NSTextField *displayField;
    IBOutlet UpOrDown *counter;
}
@end
```

Ich habe die Deklaration für die `updateDisplay:`-Methode entfernt. Wir werden diese Methode immer noch implementieren, aber sie ist nicht mehr Teil des öffentlichen Interface von `Display`.

Nun widmen wir uns dem beobachteten Objekt. Die `UpOrDown`-Header-Datei kann ein wenig geändert werden. Wir entfernen alle Referenzen auf `Display` und fügen stattdessen eine Ivar vom Typ `NSNumber` namens `count` ein.

KVO/Counter3/UpOrDown.h

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

@interface UpOrDown : NSObject {
    NSNumber *count;
}
-(IBAction) step:(id) sender;
@end
```

Wir wollen zuerst die *.nib*-Datei korrigieren und dann diese beiden Klassen implementieren. Im Interface Builder lösen wir die Verbindung auf, die zwischen dem `display`-Outlet von `UpOrDown` und `Display` besteht. Wählen Sie `Display`, und verbinden Sie seinen `counter`-Outlet mit `UpOrDown`. Speichern Sie ab.

Jetzt sind wie so weit, den Observer einrichten zu können. Das umfasst drei grundlegende Schritte:

1. Der Observer muss sich bei dem Objekt registrieren, das die zu überwachende Eigenschaft enthält.

2. Das beobachtete Objekt muss diese Eigenschaft so aktualisieren, dass Observer über die Änderung informiert werden.

3. Der Observer muss auf die empfangenen Notifikationen reagieren.

Sehen wir uns diese Schritte genauer an.

20.4 Als Observer registrieren

Die Registrierung eines Observers ist einfach. Sie müssen nur die folgende Nachricht an das zu überwachende Objekt senden:

```
addObserver:forKeyPath:options:context:
```

Übergeben Sie einen Zeiger auf den Observer als ersten Parameter. In unserem Beispiel übergeben wir `self`, Sie können aber ein beliebiges Objekt übergeben. Der Schlüsselpfad ist der Pfad zu der Eigenschaft, die Sie interessiert. Das ist die gleiche Form, die Sie gesehen haben, als Sie KVC kennengelernt haben. Vier Optionen stehen zur Verfügung, die über | kombiniert werden können.

Die vier `NSKeyValueObservingOptions` sind `NSKeyValueObservingOptionNew`, `NSKeyValueObservingOptionOld`, `NSKeyValueObservingOptionInitial` und `NSKeyValueObservingOptionPrior`. Die ersten beiden legen fest, ob Sie den neuen oder alten Wert des überwachten Attributs erhalten. Die dritte sendet eine Notifikation, wenn Sie den Observer erstmals einrichten. Die letzte legt fest, dass Sie vor und nach der jeder Änderung eine Nachricht empfangen wollen (und nicht nur danach).

Wir registrieren unser `Display` so, dass es auf Änderungen der `count`-Eigenschaft des `UpOrDown`-Objekts reagiert. Die Registrierung erfolgt, wenn `Display` aus dem Nib aufwacht.

KVO/Counter3/Display.m

```
-(void) awakeFromNib {
    [counter addObserver:self
              forKeyPath:@"count"
                 options:NSKeyValueObservingOptionNew
                 context:NULL];
}
```

Alle Objekte erben `addObserver:forKeyPath:options:context:` von `NSObject`, das `Display`-Objekt kann diese Nachricht also an das `UpOrDown`-Objekt senden, um sich als Observer für die `count`-Eigenschaft zu registrieren.

Vergessen Sie nicht, hinter sich aufzuräumen. Sie müssen die Registrierung aufheben, wenn der Observer freigegeben wird. Als wir den Speicher per Referenzzähler verwaltet haben, wurden Ressourcen über die `dealloc`-Methode freigegeben. Mit aktiver automatischer Garbage Collection verwenden wir stattdessen die `nalize`-Methode, da `dealloc` nie aufgerufen wird.

KVO/Counter3/Display.m

```
-(void) finalize {
     [counter removeObserver:self forKeyPath:@"count"];
     [super finalize];
}
```

Mehr ist zur Registrierung eines Observers nicht notwendig.

20.5 Änderungen observierbar machen

Wir haben bei Cocoa so viel Zauberei erlebt, das man glauben könnte, man bräuchte einfach nur den Wert der Ivar `count` von `UpOrDown` zu setzen, und schon würde die Änderung von `Display` aufgegriffen.

```
-(void) step: (id) sender {
    //das reicht nicht
    count = [NSNumber numberWithInteger:[sender integerValue]];
}
```

Wenn Sie eine Variable direkt setzen, müssen Sie die Änderung aber leider anzeigen, indem Sie signalisieren, dass sich der Wert ändern wird, und dann gleich noch einmal, sobald der Wert geändert wurde.

KVO/Counter3/UpOrDown.m

```
-(IBAction) step: (id) sender {
►    [self willChangeValueForKey:@"count"];
     count = [NSNumber numberWithInteger:[sender integerValue]];
►    [self didChangeValueForKey:@"count"];
}
```

Das kann einem schon Kopfschmerzen bereiten. Es gibt zwei weniger schmerzhafte Lösungen. Die eine ist Key Value Coding. Wenn Sie `setValue:forKey:` verwenden, werden die Observer benachrichtigt. KVC und KVO arbeiten zusammen. Für den Einsatz von KVC würden Sie `UpOrDown` wie folgt ändern:

KVO/Counter4/UpOrDown.m

```
-(IBAction) step: (id) sender {
►    [self setValue:[NSNumber numberWithInteger:[sender integerValue]]
►            forKey:@"count"];
}
```

Vielleicht ging Ihnen das zu schnell, um zu bemerken, wie raffiniert es war: `count` ist ein Instanzvariable und keine Eigenschaft, und wir haben einfach KVC verwendet, um ihren Wert zu ändern. Ja, ich habe das im vorigen Kapitel erwähnt, aber jetzt sehen Sie auch den zusätzlichen Nutzen. Sie ändern den Wert der Variablen und senden Notifikationen dieser Änderung mit KVO an registrierte Observer.

Eine andere Lösung ist die Deklaration und Verwendung einer Eigenschaft für die Variable `count`. Fügen Sie die Eigenschaft in die Header-Datei ein:

KVO/Counter5/UpOrDown.h

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

@interface UpOrDown : NSObject {
    NSNumber *count; }

► @property(copy) NSNumber *count;
-(IBAction) step:(id) sender;
@end
```

Setzen Sie den Eigenschaftswert direkt. Der Observer wird über die Änderung informiert:

KVO/Counter5/UpOrDown.m

```
#import "UpOrDown.h"

@implementation UpOrDown
► @synthesize count;
-(IBAction) step: (id) sender {
►     self.count = [NSNumber numberWithInteger:[sender integerValue]];
}
@end
```

In diesem Abschnitt haben Sie drei Methoden kennengelernt, um den Wert einer Variablen so zu ändern, dass Observer es mitbekommen. Erstens können sie den Wert der Variablen direkt ändern, aber dann müssen Sie auch `willChangeValueForKey:` und `didChangeValueForKey:` aufrufen. Zweitens können Sie KVC nutzen, um den Wert der Variablen über `setValue:ForKey:` zu setzen. Und drittens können Sie eine Eigenschaft einführen und anstelle der zugrunde liegenden Instanzvariablen den Wert dieser Eigenschaft ändern.

20.6 Die Änderungen überwachen

Der Observer ist nun registriert, und das Attribut ist so eingerichtet, dass es überwacht wird und den Observer über Änderungen informiert. Der letzte Schritt besteht darin, dass der Observer auf die Änderung reagiert. Ich muss Sie warnen: Das ist der Teil, über den es die meisten Klagen gibt. In unserer Anwendung könnte das wie folgt aussehen:

KVO/Counter5/Display.m

```
1  - (void)observeValueForKeyPath:(NSString *)keyPath
2                        ofObject:(id)object
3                          change:(NSDictionary *)change
4                         context:(void *)context {
5     [self updateDisplay:[object valueForKeyPath:keyPath]];
6  }
```

Die Signatur der Methode erstreckt sich über vier Zeilen, während der Rumpf nur eine einzige Zeile enthält. In Zeile 5 aktualisiere ich das Display mit dem Wert, den ich aus dem `change`-Dictionary abgerufen habe.

Was, bitteschön, gibt es da zu klagen?

Diese Methode wird für jede Eigenschaft aufgerufen, die dieses Objekt überwacht. Sie können mehrere Eigenschaften desselben Objekts überwachen, oder Änderungen in verschiedenen Objekten. Wenn Sie sich registriert haben, um über Updates einer Eigenschaft informiert zu werden, dann ist das die Methode, die aufgerufen wird, und es ist an Ihnen, herauszufinden, wer sie aufgerufen hat und was geschehen soll.

Das ist keine große Sache, unterscheidet sich aber von der Art und Weise, wie Notifikationen arbeiten. Bei Notifikationen müssen Sie festlegen, welche Methode für eine bestimmte Notifikation in welchem Objekt aufgerufen werden soll. Hier handelt es sich um einen ganz anderen Mechanismus, und diejenigen, die ihn wie Notifikationen nutzen wollen, beschweren sich lauthals.

Es gibt viele Möglichkeiten, mit dieser Einschränkung umzugehen. Eine besteht darin, sicherzustellen, dass der Schlüsselpfad auf die `count`-Eigenschaft zeigt. Aber das kann schnell hässlich und unhandlich werden, wenn man Änderungen in mehreren Eigenschaften überwacht. Der einfachste Weg, die Einschränkung zu umgehen, besteht darin, kleine, konzentrierte Observer zu entwickeln, die Änderungen an nur einer einzelnen Eigenschaft beobachten. Wir kommen auf diesen Ansatz später in diesem Kapitel zurück.

20.7 Übung: Einen zweiten Observer einfügen

Ein Vorteil von KVO besteht darin, dass das Objekt, das das zu überwachende Attribut enthält, nichts über den Observer wissen muss. Sie können so viele Observer hinzufügen, wie Sie wollen.

Genau das wollen wir tun. Fügen Sie einen zweiten Observer für die `count`-Eigenschaft in `UpOrDown` ein. Legen Sie eine Klasse namens `Logger` an, die den neuen Wert des Zählers in der Konsole ausgibt. Sie sollten keine Änderungen an `UpOrDown` oder `Display` vornehmen müssen.

20.8 Lösung: Einen zweiten Observer einfügen

Nehmen Sie eine neue Objective-C-Klasse namens `Logger` in Ihr Projekt auf. `Logger` benötigt ein Outlet, um die Verbindung mit der `UpOrDown`-Instanz herzustellen, Ihr Header muss deshalb wie folgt aussehen:

KVO/Counter6/Logger.h

```
#import <Cocoa/Cocoa.h>
@class UpOrDown;

@interface Logger : NSObject {
    IBOutlet UpOrDown *counter;
}
@end
```

Fügen Sie ein Outlet für ihr `Logger`-Objekt in `CounterAppDelegate.h` ein. Erzeugen Sie eine `Logger`-Instanz im Interface Builder und verbinden sie dessen `counter`-Outlet mit `UpOrDown`. Verbinden Sie das `logger`-Outlet von `CounterAppDelegate` mit Ihrem `Logger`-Outlet. Speichern Sie ihre Änderungen.

Die Implementierung von `Logger` ist nahezu identisch mit `Display`. Die Unterschiede sind nachfolgend entsprechend hervorgehoben. Ich zeige Ihnen den kompletten `Logger`, da Sie `Display` nur häppchenweise gesehen haben.

KVO/Counter6/Logger.m

```
#import "Logger.h"


@implementation Logger
► -(void)logValue: (NSNumber *) value{
►     NSLog(@"%@", value);
► }
-(void) awakeFromNib {
     [counter addObserver:self
                  forKeyPath:@"count"
```

```
                        options:NSKeyValueObservingOptionNew
                        context:NULL];
}
-(void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                      context:(void *)context {
►     [self logValue:[object valueForKeyPath:keyPath]];
}

-(void) finalize {
    [counter removeObserver:self forKeyPath:@"count"];
    [super finalize];
}
@end
```

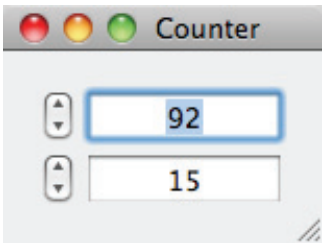
Wie Sie sehen, ist es ganz leicht, beliebig viele Observer hinzuzufügen. Sie beeinflussen das beobachtete Objekt nicht und müssen nichts über andere Observer wissen.

Andererseits wird die Sachlage komplizierter, wenn ein Objekt mehr als ein Attribut aus einem oder mehreren Objekten überwacht.

20.9 Die unschöne Lösung

Wir wollen unseren Spaß mit zwei Steppern und zwei Displays verdoppeln. Jedes Display ist mit jeweils einem Stepper verbunden. `UpOrDown` soll mit beiden Steppern interagieren. `Display` wird als Observer für den aktuellen Zählerstand beider Stepper fungieren.

Ein Objekt dient also als Observer für zwei verschiedene Attribute. Wir werden das auf zwei Arten handhaben, aber zunächst müssen wir einige Vorbereitungen treffen.



Entfernen Sie die `Logger`-Dateien. Sie stören nicht weiter, machen unser Beispiel aber ein wenig unübersichtlich. Sie müssen `Logger.h` und `Logger.m` löschen. Vergessen Sie nicht, das `Logger`-Objekt aus dem Dokumentenfenster des Interface Builders zu entfernen, und die `logger`-Eigenschaft aus `CounterAppDelegate`.

UpOrDown benötigt zwei Outlets und zwei Eigenschaften:

KVO/Counter7/UpOrDown.h

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

@interface UpOrDown : NSObject {
    NSNumber *countOne, *countTwo;
}
@property(copy) NSNumber *countOne, *countTwo;
-(IBAction) stepOne:(id) sender;
-(IBAction) stepTwo:(id) sender;
@end
```

In UpOrDown.m müssen Sie beide Variablen synthetisieren und die Aktionen implementieren, die die Eigenschaften entsprechend den aktuellen Zählerwerten setzen:

KVO/Counter7/UpOrDown.m

```
#import "UpOrDown.h"

@implementation UpOrDown

@synthesize countOne, countTwo;

-(IBAction) stepOne: (id) sender {
    self.countOne = [NSNumber numberWithInteger:[sender integerValue]];
}
-(IBAction) stepTwo: (id) sender {
    self.countTwo = [NSNumber numberWithInteger:[sender integerValue]];
}
@end
```

Fügen Sie ein zweites Outlet für das zusätzliche Textfeld in Display.h ein:

KVO/Counter7/Display.h

```
#import <Cocoa/Cocoa.h>
@class UpOrDown;

@interface Display : NSObject {
    IBOutlet NSTextField *displayFieldOne, *displayFieldTwo;
    IBOutlet UpOrDown *counter;
}
@end
```

Verbinden Sie Aktionen und Outlets im Interface Builder und löschen Sie veraltete Links.

In Xcode beenden Sie ihre Arbeit an Display.m, wobei Sie größtenteils alles doppelt machen. Wir hätten das schlau per KVC erledigen können, aber das wäre dem Kernpunkt dieses Beispiels zuwider gelaufen: wie ein Objekt mehrere Attribute überwacht.

Das Display-Objekt überwacht Änderungen sowohl an countOne als auch an countTwo. Ändert sich einer dieser Werte, wird die Methode observeValueForKeyPath: aufgerufen. Das bedeutet, dass wir herausfinden müssen, warum diese Methode aufgerufen wurde. Die hier vorgestellte Lösung untersucht den Schlüsselpfad und bei Bedarf das überwachte Objekt, um herauszufinden, wer die Notifikation angestoßen hat und warum.

KVO/Counter7/Display.m

```
#import "Display.h"

@implementation Display

-(void) updateDisplayOne: (NSNumber *) newValue{
     [displayFieldOne setIntegerValue:[newValue integerValue]];
}
-(void) updateDisplayTwo: (NSNumber *) newValue{
     [displayFieldTwo setIntegerValue:[newValue integerValue]];
}
-(void) awakeFromNib {
     [counter addObserver:self
                forKeyPath:@"countOne"
                   options:NSKeyValueObservingOptionNew
                   context:NULL];
     [counter addObserver:self
                forKeyPath:@"countTwo"
                   options:NSKeyValueObservingOptionNew
                   context:NULL];
}
-(void)observeValueForKeyPath:(NSString *)keyPath
                     ofObject:(id)object
                       change:(NSDictionary *)change
                      context:(void *)context {
     if ([keyPath isEqualToString:@"countOne"]){
          [self updateDisplayOne:[object valueForKeyPath:keyPath]];
     } else if ([keyPath isEqualToString:@"countTwo"]){
          [self updateDisplayTwo:[object valueForKeyPath:keyPath]] ;
     }
}
-(void) finalize {
     [counter removeObserver:self forKeyPath:@"countOne"];
     [counter removeObserver:self forKeyPath:@"countTwo"];
     [super finalize];
}
@end
```

Ich bin nicht besonders stolz auf Code wie diesen. Sie haben gesehen, dass wir bei der Programmierung von Cocoa mit Objective-C einen Großteil der Bedingungsanweisungen umgehen können. Das bietet uns einen sauberen Weg durch den Code.

In den nächsten beiden Abschnitten werde ich Ihnen zwei andere Lösungen vorstellen. Die erste verwendet KVC, um die richtige Methode abhängig vom Schlüssel der uns gesendeten Notifikation auszuwählen. In der zweiten führe ich dedizierte Observer-Objekte ein. Der Unterschied zwischen diesen beiden Ansätzen lässt sich im Prinzip mit dem Zeitpunkt der Entscheidung zusammenfassen: ob man sich als Observer registriert oder ob man die Notifikation empfängt. Ich bevorzuge dedizierte Observer, aber ich bin in dieser Hinsicht kein Eiferer.

20.10 Methoden wählen mit KVC

Wenn wir eine Notifikation empfangen, hat sich entweder `countOne` oder `countTwo` geändert. Eine mögliche Lösung wäre also, die Methoden für die Aktualisierung der beiden Displays zu ändern:

KVO/Counter8/Display.m

```
-(void) updateDisplayForcountOne: (NSNumber *) newValue{
    [displayFieldOne setIntegerValue:[newValue integerValue]];
}
-(void) updateDisplayForcountTwo: (NSNumber *) newValue{
    [displayFieldTwo setIntegerValue:[newValue integerValue]];
}
```

Dann könnten wir die richtige Methode wählen, indem wir einfach `countOne` oder `countTwo` an das Ende von `updateDisplayFor` anhängen und den Wert des entsprechenden Zählers als Parameter übergeben:

KVO/Counter8/Display.m

```
-(void)observeValueForKeyPath:(NSString *)keyPath
                     ofObject:(id)object
                       change:(NSDictionary *)change
                      context:(void *)context {
►    [self performSelector:NSSelectorFromString(
►             [NSString stringWithFormat:@"updateDisplayFor%@:",keyPath])
►                     withObject: [object valueForKeyPath:keyPath]];
}
```

Code wie dieser ist mir immer ein wenig zu raffiniert. Er ist clever, er funktioniert und er ist ein schönes Beispiel für den Einsatz von KVC, aber ich glaube nicht, dass der Zweck des Codes deutlich wird. Lassen Sie uns die alten Methodennamen `updateDisplayOne:` und `updateDisplayTwo:` wiederherstellen und unabhängige Observer einrichten.

20.11 Ein Observer-Objekt implementieren

Wir können sehr leicht einen `Observer` als Hilfsobjekt entwickeln. Wir richten ihn ein und teilen ihm mit, welche Methode aufgerufen werden soll, wenn eine Notifikation gesendet wird.

Legen Sie eine `Observer`-Klasse an und fügen Sie Eigenschaften ein, die Zeiger auf das Zielobjekt und die Aktion aufnehmen. Deklarieren Sie außerdem eine spezielle `init`-Methode, die es Ihnen erlaubt, diese Informationen zu übergeben.

KVO/Counter9/Observer.h

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

@interface Observer : NSObject {
    id targetObject;
    SEL targetAction;
}
@property id targetObject;
@property SEL targetAction;
-(id) initWithTarget:(id)object action: (SEL)action;

@end
```

Die Methode `observeValueForKeyPath:ofObject:change:context:` ruft die Aktion auf, die wir in der `targetAction`-Eigenschaft festgehalten haben, und wird für das Objekt aufgerufen, das in der `targetObject`-Eigenschaft steht. Sie übergibt den aktuellen Wert des Zählers als Parameter.

```
#import "Observer.h"

@implementation Observer

@synthesize targetObject, targetAction;

- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {
►    [self.targetObject performSelector:self.targetAction
►                            withObject:[object valueForKeyPath:keyPath]];
}
-(id) initWithTarget:(id)object action: (SEL)action {
    if (self =[super init]) {
        self.targetObject = object;
        self.targetAction = action;
    }
    return self;
}
@end
```

Nun können wir die Display-Klasse vereinfachen. Deklarieren Sie die Instanzvariablen observerOne und observerTwo vom Typ Observer in der Display-Header-Datei:

KVO/Counter9/Display.h

```
#import <Cocoa/Cocoa.h>
@class UpOrDown;
► @class Observer;

@interface Display : NSObject {
    IBOutlet NSTextField *displayFieldOne, *displayFieldTwo;
    IBOutlet UpOrDown *counter;
►   Observer *observerOne, *observerTwo;
}
@end
```

Registrieren Sie die neuen Observer in awakeFromNib. Entfernen Sie die observeValueForKeyPath:ofObject:change:context:-Methode. Sie ist jetzt im Observer implementiert. Geben Sie die Observer in der nalize-Methode frei.

KVO/Counter9/Display.m

```
#import "Display.h"
#import "Observer.h"

@implementation Display

-(void) updateDisplayOne: (NSNumber *) newValue{
     [displayFieldOne setIntegerValue:[newValue integerValue]];
}

-(void) updateDisplayTwo: (NSNumber *) newValue{
     [displayFieldTwo setIntegerValue:[newValue integerValue]];
}

-(void) awakeFromNib {
    observerOne = [[Observer alloc]
                   initWithTarget:self
                           action:@selector(updateDisplayOne:)];
    observerTwo = [[Observer alloc]
                   initWithTarget:self
                           action:@selector(updateDisplayTwo:)];
    [counter addObserver:observerOne
              forKeyPath:@"countOne"
                 options:NSKeyValueObservingOptionNew
                 context:NULL];
    [counter addObserver:observerTwo
              forKeyPath:@"countTwo"
                 options:NSKeyValueObservingOptionNew
                 context:NULL];
}
```

```
-(void) finalize {
     [counter removeObserver:observerOne forKeyPath:@"countOne"];
     [counter removeObserver:observerTwo forKeyPath:@"countTwo"];
     [super finalize];
}
@end
```

Kleine, einem bestimmten Zweck dienende Klassen wie `Observer` machen Ihren Code übersichtlicher. Jede Instanz von `Observer` weiß, welchem Objekt sie welche Nachricht senden soll, wenn sie aufgerufen wird.

20.12 Abhängige Variablen aktualisieren

Manchmal finden sich in einer Klasse Variablen, die vom Wert anderer Variablen abhängig sind. Diese abhängigen Variablen können mithilfe der unabhängigen Variablen immer wieder neu berechnet werden. Nehmen wir zum Beispiel an, wir wollen eine neue Variable namens `totalCount` in `UpOrDown` einführen, die die Summe von `countOne` und `countTwo` enthält. Sobald der Benutzer einen der Stepper anklickt, ändert sich der Wert von `totalCount`.

Wir müssen die folgenden Dinge tun können

- Die `totalCount`-Abhängigkeit von `countOne` und `countTwo` registrieren.
- Einen Observer einfügen, der auf Veränderungen in `totalCount` reagiert. Tatsächlich wird er auf Änderungen aller Attribute reagieren, die wir im vorigen Schritt registriert haben.
- Festlegen, was zu tun ist, wenn wir eine Notifikation über eine Änderung in `totalCount` empfangen.

Es gibt zwei Arten zum Registrieren von `totalCount`-Abhängigkeiten. Eine dieser Methoden ist:

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
```

Diese Klasse kann verwendet werden, um Abhängigkeiten für eine beliebige Anzahl von Variablen zu registrieren. Sie können verschiedene `NSSets` für jeden Schlüssel zurückgeben. Eine alternative Form der Methode hängt den Variablennamen an das Ende von `keyPathsForValuesAffecting` an. Für die Variable `totalCount` verwenden wir also die folgende Methode, um die Abhängigkeit von `countOne` und `countTwo` zu registrieren:

```
+(NSSet *)keyPathsForValuesAffectingTotalCount {
    return [NSSet setWithObjects:@"countOne",@"countTwo",nil];
}
```

In der Header-Datei für UpOrDown fügen Sie eine Deklaration für einen Zeiger auf eine NSNumber namens totalCount ein:

KVO/Counter10/UpOrDown.h

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

@interface UpOrDown : NSObject {
    NSNumber *countOne, *countTwo, *totalCount;
}

@property(copy) NSNumber *countOne, *countTwo, *totalCount;
-(IBAction) stepOne:(id) sender;
-(IBAction) stepTwo:(id) sender;
@end
```

Hier die Implementierung von UpOrDown:

KVO/Counter10/UpOrDown.m

```
#import "UpOrDown.h"

@implementation UpOrDown

@synthesize countOne, countTwo, totalCount;
-(IBAction) stepOne: (id) sender {
    self.countOne = [NSNumber numberWithInteger:[sender integerValue]];
}
-(IBAction) stepTwo: (id) sender {
    self.countTwo = [NSNumber numberWithInteger:[sender integerValue]];
}
+(NSSet *)keyPathsForValuesAffectingTotalCount {
    return [NSSet setWithObjects:@"countOne",@"countTwo", nil];
}
-(NSNumber *) totalCount {
    return [NSNumber numberWithInt:
         [self.countOne intValue] + [self.countTwo intValue]];
}
-(void)awakeFromNib{
     [self addObserver:self
             forKeyPath:@"totalCount"
                 options:NSKeyValueObservingOptionNew
                 context:NULL];
}
-(void)observeValueForKeyPath:(NSString *)keyPath
                       ofObject:(id)object
                         change:(NSDictionary *)change
                        context:(void *)context {
    NSLog(@"%@", self.totalCount);
}
@end
```

Neben `keyPathsForValuesAffectingTotalCount:` registrieren wir den Observer in der `awakeFromNib`-Methode. Wir verwenden außerdem eine Getter-Methode für `totalCount`, die uns die Summe aus `countOne` und `countTwo` zurückgibt. Diese wird von `observeValueForKeyPath:ofObject:change:context:` aufgerufen, wenn sich einer der Werte ändert. Wir geben diese Summe einfach in der Konsole aus.[1]

Dieses Kapitel hat Ihnen die Observer-Grundlagen vermittelt.[2] Sie können ein Objekt registrieren und auf einen Aufruf warten. Das ist wie eine Notifikation ohne Notication Center. Es handelt sich einfach um eine Beziehung zwischen zwei Objekten, wobei nur der Observer etwas über den Überwachten wissen muss. Sie haben gesehen, wie man mehrere Observer mit einem Attribut und einen Observer mit mehreren Attributen registriert. Schließlich haben Sie gesehen, wie man eine abhängige Variable so einrichtet, dass ihre Observer aktualisiert werden, sobald sich die Variablen ändern, von denen sie abhängig ist.

So cool KVO und KVC für sich genommen und in Verbindung miteinander auch sind, ihre eigentliche Stärke liegt in den Techniken, die sie ermöglichen. Wir werden Code aus der Controller-Schicht durch Bindungen und in der Modellschicht durch Core Data ersetzen.

Sie können KVO und KVC sowohl für die Mac OS X- als auch für die iPhone OS-Entwicklung benutzen. Wenn Sie hauptsächlich für Mac OS X entwickeln, wird Ihnen das, was gleich kommt, richtig gut gefallen. Sie werden sehen, wie diese Techniken Bindungen ermöglichen, die es Ihnen erlauben, mit weniger Code in der Controller-Schicht auszukommen. Im iPhone OS werden Bindungen momentan leider nicht unterstützt.

1 `KVO/Counter11` in der Code-ZIP-Datei zeigt Ihnen, wie Sie die Instanzvariable und Eigenschaft `totalCount` entfernen und dennoch das gleiche Verhalten erreichen können.

2 Weitere Informationen finden Sie in Apples *Key-Value Observing Guide* [App08f].

Kapitel 21

Cocoa-Bindungen

Bisher haben wir den Interface Builder benutzt, um unsere GUI aufzubauen und Buttons, Labels, Tabellen und so weiter mit den Controllern und Modellen zu verbinden. Dadurch erübrigt sich eine Menge des Standardcodes. Sie müssen im Programmcode nicht von `NSButton` erben und dessen Aussehen und Platzierung konfigurieren. Durch unsere Arbeit mit eigenen Views wissen Sie, dass das programmtechnisch möglich ist – es ist aber einfach nicht notwendig.

Cocoa-Bindungen sehen sich die Controller-Schicht an und fragen sich, wie viel vom sich wiederholenden Code entfernt werden kann. Wenn man sorgfältig mit KVO- und KVC-konformen Klassen arbeitet, lautet die Antwort „der größte Teil". In diesem Kapitel werden wir uns hauptsächlich im Interface Builder aufhalten, Verbindungen herstellen und Schlüsselpfade eingeben. Wir müssen ein wenig Code entwickeln, aber wirklich nicht viel.

Wir beginnen dieses Kapitel mit der Entwicklung einer neuen Cocoa-Anwendung namens *CounterWithBindings*. Sie werden eine neue Objective-C-Klasse namens `Counter` anlegen und eine Eigenschaft namens `count` vom Typ `NSNumber` deklarieren und synthetisieren. Nur dieser Code ist notwendig, um Ihr Modell entwickeln zu können.

Für den View ziehen Sie einen Stepper und ein Textfeld in das Fenster. Wir werden im Verlauf des Kapitels weitere hinzufügen, aber für den Anfang reicht das aus.

Bleibt noch der Controller. In diesem Kapitel werden Sie den Controller nicht in Programmcode implementieren. Sie werden Cocoa-Bindungen und die Tatsache, dass Ihr Modell KVC- und KVO-konform ist, nutzen, um Ihren Controller im Interface Builder zu erzeugen. Cocoa-Bindungen

lassen den View und das Modell kommunizieren, ohne dass wir allzu viel Standard-Glue-Code schreiben müssten. Wir wollen damit beginnen, uns das Beispiel aus dem vorigen Kapitel noch einmal anzusehen.

21.1 Modell und View für unseren Zähler mit Bindungen

Legen Sie ein neues Projekt namens *CounterWithBindings* an. Fügen Sie eine neue Objective-C-Klasse namens `Counter` ein, die eine Eigenschaft namens `count` besitzt. Fügen Sie mit anderen Worten die beiden folgenden Codezeilen in die Header-Datei-Vorlage ein:

Bindings/CounterWithBindings1/Counter.h

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

@interface Counter : NSObject {
    NSNumber *count;
}
@property(copy) NSNumber *count;

@end
```

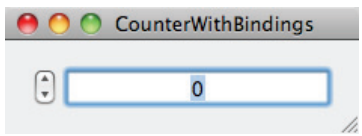
Sie müssen außerdem eine Zeile Code in die Implementierungsdatei einfügen, um die Eigenschaft `count` zu synthetisieren:

Bindings/CounterWithBindings1/Counter.m

```
#import "Counter.h"

@implementation Counter

@synthesize count;

@end
```

Wir haben eine einzige Klasse mit einer einzigen Eigenschaft definiert. Im ersten Teil dieses Kapitels wird es keinen weiteren Programmcode für diese Anwendung geben.

Nun wollen wir den View entwickeln. Ziehen Sie einen Stepper und ein Textfeld in das Hauptfenster. Wie vorhin verwenden Sie den Attributes Inspector, um das Verhalten des Steppers mit Value Wraps festzulegen. Passen Sie die Größe des Fensters so an, dass es wie folgt aussieht:



Okay. Wir haben nun ein Modell und einen View. Lassen Sie uns den Controller entwickeln.

21.2 Den NSObjectController aufbauen und verknüpfen

Im Interface Builder ziehen Sie ein `Counter`-Objekt aus der Library in das Dokumentenfenster. Das ist die Instanz Ihres Modells.

Joe fragt...

Wäre Code nicht besser?

Abhängig vom persönlichen Hintergrund kann es einem so vorkommen. Ein Großteil der Logik wird jetzt in Dateien verschoben, die Sie im Interface Builder einrichten. Wenn Sie ein Problem haben, können Sie nicht auf die Techniken zurückgreifen, die Sie sonst benutzt haben. Anfangs ist es schwieriger, Probleme zu lokalisieren. Andererseits werden Sie wahrscheinlich weit weniger Probleme haben, die sich durch Fehler im Programmcode einschleichen.

Cocoa-Bindungen können furchteinflößend sein. Doch Sie werden sehr schnell ein Gefühl dafür entwickeln, wo Sie nachschauen müssen, wenn etwas schiefgeht. Wenn Sie an Ihre Anfänge als Entwickler zurückdenken, dann war das damals auch so. Sie mussten lernen, wo man nach Problemen suchen muss. Sie haben wahrscheinlich lange gesucht, bis Sie merkten, dass Sie = eingegeben, aber == gemeint hatten. Es geht also nicht um neuen Frust, sondern eher darum, eine alte Grenze durch eine neue zu ersetzen.

Als Nächstes ziehen Sie einen `NSObjectController` aus der Library in das Dokumentenfenster. Erinnern Sie sich an die ersten Beispiele in diesem Buch. Der Controller sitzt zwischen dem Modell und dem View. Der Controller besteht darauf, der Vermittler zu sein. Sie müssen Ihren Controller in drei Schritten konfigurieren:

1. Die Verbindung zu dem Objekt herstellen, das der Objekt-Controller kontrolliert.
2. Festlegen, für welche Schlüssel der Objekt-Controller verantwortlich ist.
3. Jedes Element an eine Eigenschaft binden, die der Objekt-Controller kontrolliert.

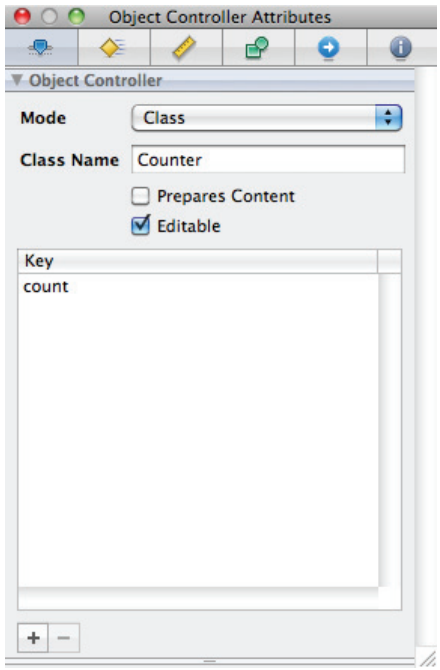
Den Controller verbinden

Der Objekt-Controller muss wissen, welches Objekt er kontrolliert. Klicken Sie ihn an und suchen Sie im Connections Inspector nach seinem `content`-Outlet. Verknüpfen Sie diese `content`-Verbindung mit dem `Counter`-Objekt. Es kann (und wird) mehr als ein Objekt vom gleichen Typ geben. Das ist der Schritt, in dem Sie festlegen, für welches davon der Objekt-Controller verantwortlich ist.

Die Schlüssel wählen

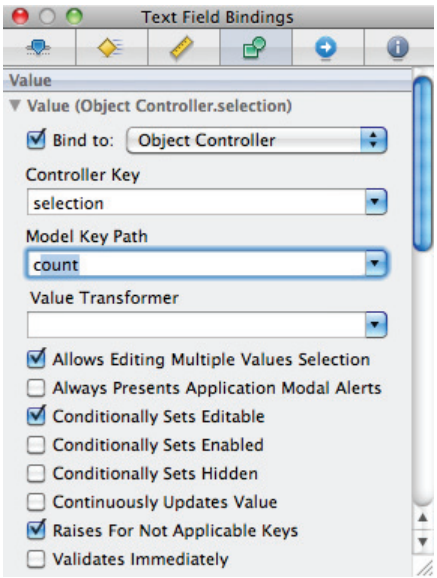
Als Nächstes müssen Sie denjenigen Objekten den `count`-Schlüssel zugänglich machen, die sich an den `NSObjectController` binden. Wählen Sie den Objekt-Controller und verwenden Sie den Attributes Inspector, um den Klassennamen mit `Counter` anzugeben. Beachten Sie, dass es zwei Modi gibt. Wir verwenden den Klassenmodus, weil `Counter` eine Klasse ist. Wenn wir Datenmodelle mit Core Data aufbauen, wählen wir den Entitätenmodus.

Sie müssen explizit festlegen, welche Schlüssel der Objekt-Controller für Bindungen zur Verfügung stellt. Hier ist nur eine Wahl möglich. Verwenden Sie das Pluszeichen, um den Schlüssel `count` hinzuzufügen.



Eine Komponente binden

Wählen Sie das Textfeld durch einen Klick aus und öffnen Sie den Bindings Inspector mit D4 oder durch Anklicken des vierten Reiters.



Im Value-Abschnitt wählen Sie die *Bind to* -Checkbox und benutzen die Drop-down-Liste, um den Object Controller auszuwählen. Der „Controller Key“ ist `selection`, und der „Model Key Path“ ist `count`. Tun Sie das Gleiche für den Stepper.

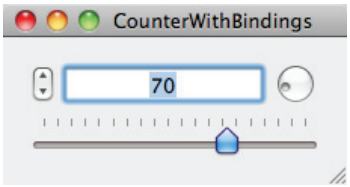
Sowohl der Stepper als auch das Textfeld werden an den Wert von `count` gebunden. Klicken Sie auf *Build & Run*. Klicken Sie den Stepper an, und Sie sehen, dass sich die Werte im Textfeld verändern. Das sieht nicht nach einer großen Sache aus, ist aber eine.

Wenn Sie den Stepper anklicken, wird sein Wert an den Wert von `count` gebunden, `count` wird also automatisch aktualisiert. In gleicher Weise ist das Textfeld an den Wert von `count` gebunden. Wenn sich der Wert von `count` ändert, wird das Textfeld automatisch aktualisiert. Die Verbindungen werden vom Controller hergestellt. Dabei passiert eine ganze Menge. Wir wollen im nächsten Abschnitt weitere Widgets hinzufügen, um das zu verdeutlichen.

21.3 Weitere Objekte binden

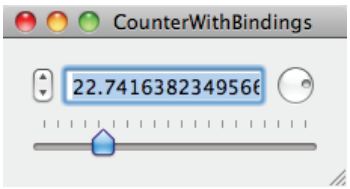
Fügen Sie einen kreisförmigen und einen horizontalen Slider in den View ein. Binden Sie beide mithilfe des Objekt-Controllers an `count`.

Ihre Slider sollten etwa so aussehen:



Wenn Sie den Stepper anklicken, werden der Wert von `count` aktualisiert und alle gebundenen Elemente über die Änderung informiert. Wenn ich also (wie in der Abbildung zu sehen) den Wert 70 in das Textfeld eintrage, bewegen sich die beiden Slider an die entsprechende Stelle.

Ziehen Sie den horizontalen Slider ganz nach rechts auf den Wert *100*. Verwenden Sie den Stepper, um den Wert um eins zu verringern. Der horizontale Slider sollte sich entsprechend nach links bewegen. Drehen Sie den kreisförmigen Slider ein wenig. Sie sollten etwas sehen wie das hier:



Ich bin nicht ganz glücklich damit, wie die Slider im Moment funktionieren. Ich möchte meinen `count` mit ganzen Zahlen arbeiten lassen, doch wie Sie sehen, liefern meine Slider zu viele Stellen rechts vom Dezimalpunkt.

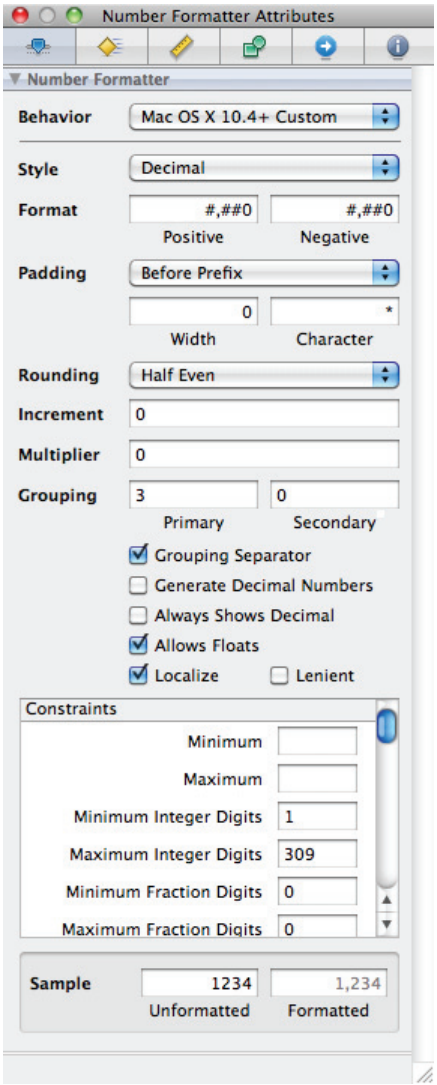
Wir könnten einen Slider wählen und im Attributes Inspector „Only stop on tick marks" („nur an den Skalenstrichen einrasten") festlegen. Ich habe meinen horizontalen Slider mit 20 Ticks versehen, ich könnte also mit einem Vielfachen von fünf arbeiten. Damit hätte ich die Sache auf ganze Zahlen beschränkt, könnte aber keine Werte auswählen, die kein Vielfaches von fünf sind.

Wir werden einen Zahlenformatierer nutzen, um das gewünschte Ziel zu erreichen.

21.4 Zahlenformatierer

Suchen Sie in der Interface Builder-Library nach einem `NSNumberFormatter`. Ziehen Sie ihn aus der Library in das Textfeld. Wenn das Textfeld ausgewählt ist, sollte das Zahlenformatierer-Icon unter dem Textfeld zu sehen sein. Möglicherweise müssen Sie zuerst ein anderes Element auswählen und dann wieder das Textfeld, damit das Icon erscheint. Sie finden den Zahlenformatierer auch im Dokumentenfenster.

Wählen Sie den Zahlenformatierer und werfen Sie dann einen Blick in den Attributes Inspector:

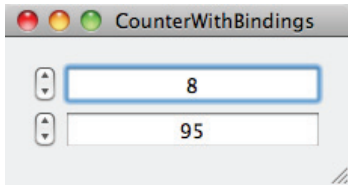


Sie können eine ganze Reihe von Parametern festlegen, von denen wir allerdings nur einige anpassen wollen. Beachten Sie die Drop-down-Liste am oberen Rand des Inspectors. Zwei Arten von Zahlenformatierung stehen Ihnen zur Verfügung. Wir arbeiten mit der moderneren Variante, also stellen Sie sicher, dass *Mac OS X 10.4+ Custom* gewählt ist. Sie können sich einen Integerwert als Dezimalzahl ohne Nachkommastellen vorstellen. Also müssen Sie zuerst den Stil auf *Decimal* festlegen. Dann wechseln Sie in den Abschnitt „Constraints" (also Einschränkungen) und legen die Nachkommastellen (Maximum Fraction Digits) auf 0 fest.

Das war's. Speichern Sie ab. Klicken Sie auf *Build & Run*, und Ihre Slider können ausschließlich Werte von 0 bis 100 annehmen.[1]

21.5 Übung: Zwei Zähler mit Bindungen verknüpfen

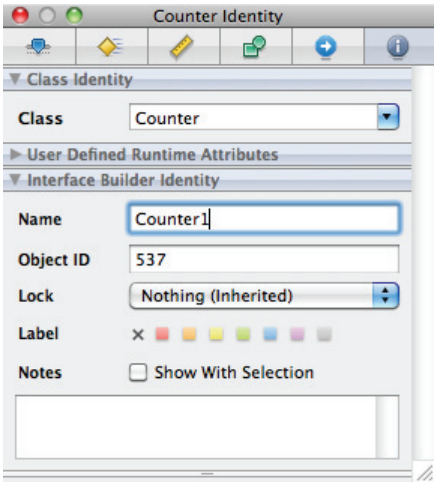
Lassen wir ein weiteres Beispiel aus dem vorigen Kapitel wiederauferstehen, ohne irgendwelchen Programmcode zu entwickeln. Die Anwendung soll zwei Stepper und zwei Textfelder besitzen. Der obere Stepper verändert nur das obere Textfeld und der untere Stepper nur das untere Textfeld.



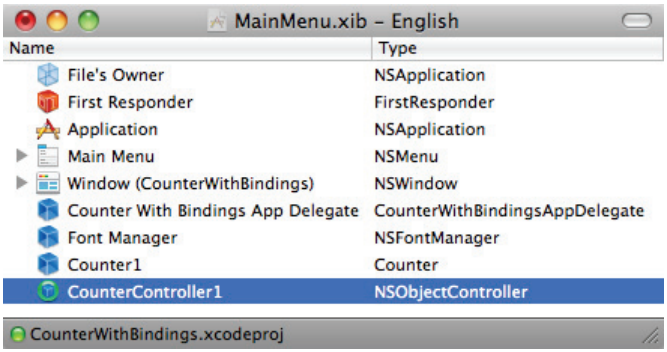
Sie sollten auch hier in der Lage sein, die Lösung vollständig im Interface Builder zu entwickeln, ohne eine Zeile Code ändern oder hinzufügen zu müssen.

Hier noch ein nützlicher Tipp: Wählen Sie den `Counter` und sehen ihn sich mit dem Identity Inspector an. Am unteren Rand sollten Sie einen Abschnitt mit dem Namen „Interface Builder Identity" sehen.

1 Diese Lösung entspricht `Bindings/CounterWithBindings3` in den Codebeispielen.



Sie können erkennen, dass ich den Namen des leeren Textfeldes in *Counter1* geändert habe. Machen Sie das auch und sehen Sie sich das Dokumentenfenster an.



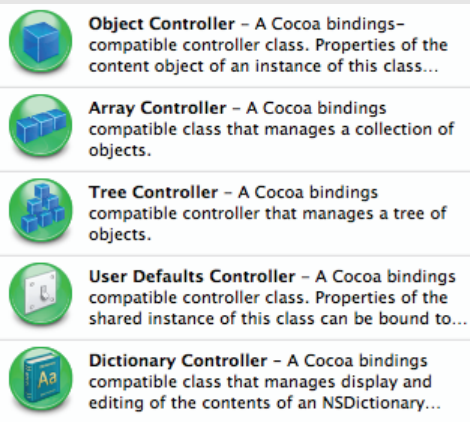
Wir haben den Namen des Objekts in *Counter1* geändert, ohne die dazugehörige Klasse zu ändern. Das macht es einfacher für Sie, die beiden Zähler zu unterscheiden. Sie können auch erkennen, dass ich den Namen von `NSObjectController` geändert habe. Wir müssen uns um die sichtbaren Elemente keine Gedanken machen, weil wir das eine oder das andere Textfeld anklicken können und wissen, welches gemeint ist. Bei den unsichtbaren Elementen ist die Möglichkeit, ihnen unterschiedliche Namen zu geben, wirklich sehr hilfreich.

21.6 Lösung: Zwei Zähler mit Bindungen verknüpfen

Sie müssen parallel zum bestehenden System noch ein zweites aufbauen. Sie werden also zwei Stepper, zwei Textfelder, zwei Controller und zwei Modellobjekte verwenden.

Die Familie der NSController

`NSObjectController` erweitert die abstrakte Klasse `NSController`. Hier ein Familienportrait von `NSObjectController` und seinen Verwandten:



Zuerst wollen wir ein wenig aufräumen. Entfernen Sie die beiden Slider aus dem Interface und fügen Sie einen weiteren Stepper und ein zusätzliches Textfeld ein. Ich habe den Zahlenformatierer entfernt, aber das ist eigentlich nicht nötig. Sie können diesen Schritt beschleunigen, indem Sie den vorhandenen Stepper und das Textfeld auswählen, mit `DD` duplizieren und die neuen Elemente an der gewünschten Stelle platzieren.

Ändern Sie im Identity Inspector die Namen von `Counter` und `NSObjectController` in `Counter1` und `CounterController1`.

Ziehen Sie ein Counter-Objekt aus der Library in das Dokumentenfenster und ändern Sie seinen Namen in `Counter2`. Ziehen Sie einen neuen Objekt-Controller aus der Library und ändern Sie seinen Namen in `CounterController2`.

Nun wiederholen Sie einfach die Schritte von vorhin. Wählen Sie `CounterController2` und verbinden Sie im Connections Inspector das `content`-Outlet mit `Counter2`. Wechseln Sie in den Attributes Inspector, legen Sie die Klasse mit `Counter` fest und fügen Sie den Schlüssel `count` hinzu.

Der letzte Schritt besteht darin, den unteren Stepper und das untere Textfeld zu binden. Wählen Sie sie nacheinander aus und nutzen Sie den Bindings Inspector, um die Bindung zu `CounterController2` herzustellen. Legen Sie `selection` als Wert des Controller Key fest und `count` für den Model Key Path.[2]

21.7 Das Modell unseres Bücherregal-Beispiels

Wir wollen uns das Bücherregal-Beispiel noch einmal ansehen. Dabei bewegen wir uns von Bindungen und Objekt-Controllern hin zu Bindungen mit Array-Controllern.

Legen Sie ein neues Cocoa-Projekt namens *BookshelfWithBindings* an und fügen Sie die `PragBook`-Klasse mit den beiden Eigenschaften `title` und `author` (beides Strings) hinzu:

Bindings/BookshelfWithBindings/PragBook.h

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

@interface PragBook : NSObject {
    NSString *author, *title;
}
@property (copy) NSString *author, *title;
@end
```

Synthetisieren Sie die Eigenschaften in der Implementierung:

Bindings/BookshelfWithBindings/PragBook.m

```
#import "PragBook.h"

@implementation PragBook
@synthesize author, title;
@end
```

Legen Sie eine `Bookshelf`-Klasse an, die ein oder mehrere `PragBook` enthalten kann. Die `Bookshelf`-Klasse enthält ein `NSMutableArray` namens `bookList` als einzige Eigenschaft:

2 Die Lösung finden Sie im Codedownload in `Bindings/CounterWithBindings4`.

Bindings/BookshelfWithBindings/Bookshelf.h

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

@interface Bookshelf : NSObject {
    NSMutableArray *bookList;
}
@property(retain) NSMutableArray *bookList;
@end
```

Wir müssen die `bookList` initialisieren, bevor wir mit ihr arbeiten können. Das geschieht in unserer `awakeFromNib`-Methode:

Bindings/BookshelfWithBindings/Bookshelf.m

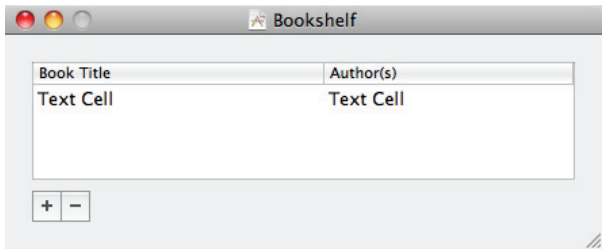
```
#import "Bookshelf.h"

@implementation Bookshelf
@synthesize bookList;
-(void) awakeFromNib {
    self.bookList = [NSMutableArray arrayWithCapacity:1];
}
@end
```

`Bookshelf` weiß nichts von `PragBook`. Wir werden später Bindungen benutzen, um festzulegen, dass `PragBook` die Art von Objekt ist, die in `bookList` enthalten ist.

Das war's. Das ist der ganze Code, den wir für dieses Beispiel entwickeln mussten.[3] Sobald Sie sehen, was unser fertiges Projekt leistet, werden Sie von der Leistungsfähigkeit von Bindungen und dem `NSArrayController` überzeugt sein.

21.8 Den View für unser Bücherregal entwickeln

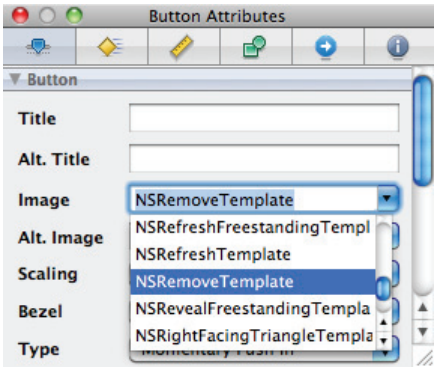
Fügen Sie einen Tabellen-View und zwei quadratische Buttons in Ihr Fenster ein und passen Sie die verschiedenen Überschriften an, damit das Ganze etwa so aussieht:



3 Das Wort *mussten* ist nicht ganz korrekt: Wir können (und werden) weniger Code schreiben, wenn wir bei Core Data angekommen sind.

Der schwierigste Teil ist vielleicht das Setzen der Plus- und Minuszeichen in den quadratischen Buttons. Im ersten Moment werden Sie wahrscheinlich einfach + oder – für die Button-Titel angeben. Das funktioniert, sieht aber nicht besonders schön aus. Apple stellt eine Reihe von Grafiken zur Verfügung, die Sie mit der Drop-down-Liste namens Image einfügen können.

Wählen Sie `NSAddTemplate` für das Plus und `NSRemoveTemplate` für das Minus:



Diesmal müssen Sie keine Identifier für die Tabellenspalten anlegen. Diese Arbeit erledigen wir mit Bindungen.

21.9 Bindung mit dem NSArrayController

Fügen Sie einen Array-Controller und ein `Bookshelf`-Objekt in Ihr Dokumentenfenster ein. Damit stehen uns alle benötigten Zutaten im Interface Builder zur Verfügung:

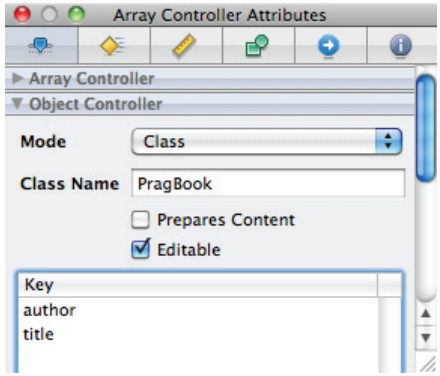
Die folgenden Schritte sind nötig, um die Bindungen aufzubauen:

1. Das im Array enthaltene Objekt festlegen sowie die Schlüssel, an die wir binden wollen.
2. Den Array-Controller an das es enthaltende Array binden.
3. Die Tabellenspalten binden.

Dann müssen wir die Plus- und Minus-Buttons verknüpfen.

Den Inhalt des Arrays festlegen

Wenn der Benutzer den Plus-Button drückt, wird ein Objekt in das `bookList`-Array eingefügt. Wir müssen festlegen, dass dieses Objekt eine Instanz von `PragBook` ist. Wählen Sie den Array-Controller und öffnen Sie den Attributes Inspector.

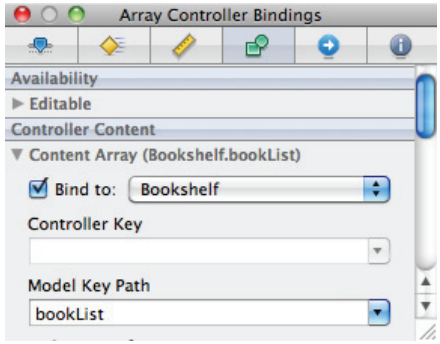


Legen Sie den Klassennamen mit `PragBook` fest. Fügen Sie die Schlüssel für `author` und `title` hinzu.

Den Controller mit seinem Array verbinden

Sie haben den Objekt-Controller über das *content*-Outlet mit seinem Objekt verbunden. Diesmal binden wir den Array-Controller mithilfe des *content*-Arrays an das Array. Wählen Sie den Array-Controller und öffnen Sie den Bindings Inspector.

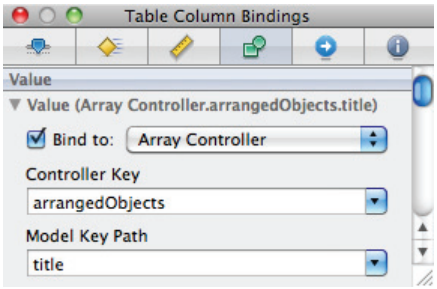
Suchen Sie unter Controller Content nach Content Array. Aktivieren Sie die „Bind to"-Checkbox und wählen Sie *Bookshelf* aus der Dropdown-Liste. Legen Sie `bookList` als Model Key Path fest.



Sie haben den Array-Controller konfiguriert. Als Nächstes binden wir die Elemente des Views an unseren Array-Controller, um die Verknüpfungen unserer Anwendung abzuschließen.

Die Tabellenspalten konfigurieren

Nun wollen wir die Tabellenspalten so binden, dass die erste Spalte mit dem Namen des Buches und die zweite mit dem Namens des Autors gefüllt wird. Wählen Sie die erste Spalte und öffnen Sie den Bindings Inspector.

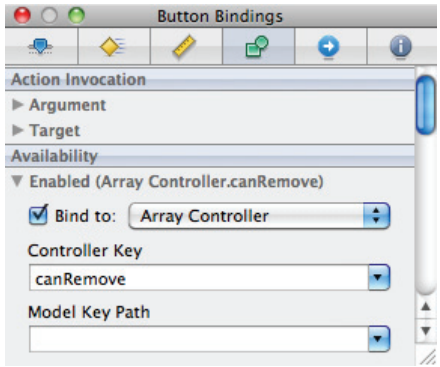


Binden Sie sie an den Array-Controller. Beachten Sie, dass der Controller Key diesmal `arrangedObjects` ist und nicht `selection`, weil die Spalte mit diesem Teil des gesamten Arrays verknüpft ist. Setzen Sie den Model Key Path auf `title`. Wiederholen Sie diesen Schritt für die zweite Spalte, aber diesmal geben Sie für Model Key Path `author` an.

Die Buttons verknüpfen

Verwenden Sie den Connections Inspector, um die Buttons zu verknüpfen. Ziehen Sie die *sent*-Aktion des Plus-Buttons in den Array-Controller. Es erscheint eine Liste möglicher Aktionen. Wählen Sie `add:`. Das Gleiche tun Sie für den Minus-Button, aber diesmal wählen Sie `remove:`.

Bevor wir fertig sind, stellen wir sicher, dass der Minus-Button nur aktiv ist, wenn Elemente vorhanden sind, die aus dem Array entfernt werden können. Wählen Sie den Minus-Button und öffnen Sie den Bindings Inspector.



Bisher haben wir nur `selection` und `arrangedObjects` für den Controller Key genutzt. Es gibt viele andere Optionen, die Situationen entsprechen, die bei der Arbeit mit Arrays auftreten. Eine davon ist `canRemove:`. Daran binden wir den Minus-Button im Bindings Inspector unter `Availability > Enabled`.

21.10 Das große Finale

Klicken Sie auf *Build & Run*. Klicken Sie den Plus-Button an und fügen Sie einige Bücher in die Liste ein. Klicken Sie den Minus-Button an, um einige zu entfernen.

Nein, das ist keine Zauberei, aber schon ziemlich nah dran. Mit minimalem Codeaufwand haben wir eine Anwendung entwickelt, die es uns erlaubt, Daten in eine Tabelle einzufügen und wieder zu enfernen.

Das sollte ausreichen, um Ihnen den Einstieg in Controller zu ermöglichen. Sie können den Controller mit den Benutzer-Standardeinstellungen benutzen, um GUI-Elemente an die gespeicherten Einstellungen zu binden. Die Dictionary- und Tree-Controller ähneln dem Array-Controller, arbeiten aber (natürlich) mit Dictionaries und Bäumen. Weitere Informationen finden Sie in Apples *Cocoa Bindings Programming Topics* [App08a] und *Cocoa Bindings Reference* [App08b].

Sie können Ihre Controller auf diese Weise auch um Code erweitern, so wie wir es für unseren View getan haben. Damit können Sie Dinge erreichen, die im Interface Builder nicht so einfach möglich sind. Ich sage nicht, dass Sie nicht programmieren sollen – ich sage nur, dass Sie die Vorteile nutzen sollten, die man Ihnen kostenlos zur Verfügung stellt, und nur dann Code entwickeln sollten, wenn Sie Code entwickeln *müssen*. Im nächsten Kapitel zu Core Data gehen wir noch einen Schritt weiter.

Kapitel 22

Core Data

Wir waren bei der Trennung von Modell, View und Controller recht erfolgreich. Wir haben bei der Arbeit an eigenen Views gesehen, dass wir Code entwickeln können, der diese Views beschreibt. Wir haben aber auch die Leistungsfähigkeit und Flexibilität gesehen, die uns der Interface Builder bei der Entwicklung unserer Views bietet. Sie haben auch gesehen, wie die Controller-Schicht das Modell und den View zusammenhält. Wir haben einen Großteil der Controller-Logik programmiert, aber Sie haben auch die Leistungsfähigkeit von Cocoa-Bindungen gesehen, die uns erneut die Verknüpfung von View und Modell im Interface Builder ermöglichen.

Doch wenn wir uns die letzten paar Kapitel ansehen, dann war es mit unseren Modellen nicht weit her. Wir haben eine Reihe von Klassen entwickelt, die ein paar Eigenschaften enthielten. Core Data erlaubt uns die Entwicklung solcher Modelle mit GUI-Tools. Entitäten und Attribute sind die Core Data-Analogie zu Klassen und Eigenschaften.

Core Data stellt Ihnen ein Werkzeug zur Verfügung, mit dem Sie die Elemente Ihres Modells und deren Interaktion miteinander entwickeln können. Es erzeugt automatisch die Klassen und Objekte, die Ihr Programm benötigt, um mit diesen Modellobjekten arbeiten zu können, und es kümmert sich um die Persistenz. Wenn Sie in Ihrer Cocoa-App Daten auf die Festplatte schreiben oder von der Festplatte lesen, sollten Sie mit Core Data arbeiten.

In diesem Kapitel sehen wir uns das Buch-Beispiel noch einmal an und entwickeln ein Entity-Relationship-Modell mit den Xcode-Tools zur Datenmodellierung. Wir werden das gesamte Beispiel neu entwickeln und erweitern, ohne eine einzige Zeile Code zu schreiben. Jedes Buch wird einen Titel, ein oder mehrere Autoren und eine Reihe von Kapiteln besitzen. Wir erlauben dem Benutzer, diese Informationen für jedes gewählte Buch einzufügen, zu löschen und zu verändern. Wir werden uns auch ansehen, wie Bindungen und Core Data zusammenarbeiten, um diese Tabellen zu füllen, wie man die Informationen auf der Festplatte speichert und wieder abruft, und wie man Sortierung und Suche integriert.

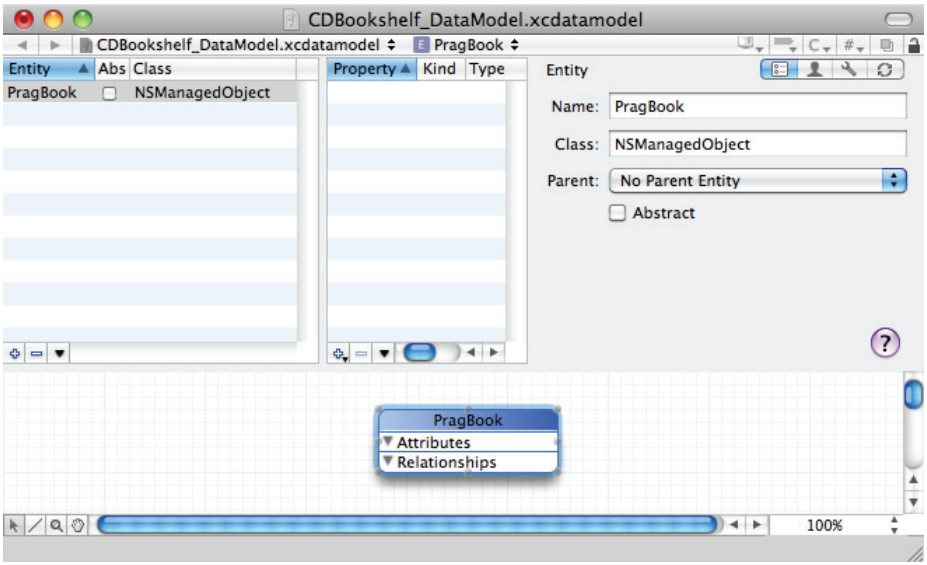
22.1 Entitäten und Attribute

Wir wollen damit beginnen, das Modell unseres Bücherregals mit Core Data neu aufzubauen. Blättern Sie ein paar Seiten zurück zu Abschnitt 21.7, *Das Modell unseres Bücherregal-Beispiels*, auf Seite 353. An diesem Modell ist nicht viel dran. Die `PragBook`-Header-Datei deklariert zwei Eigenschaften namens `author` und `title`, und die Implementierung synthetisiert deren Akzessormethoden.

Statt mit einer Klasse namens `PragBook` arbeiten wir jetzt mit Begriffen aus der Datenbankwelt. Wir legen jetzt also in unserem Datenmodell eine Entität namens `PragBook` an. Wir arbeiten in unserem Datenmodell außerdem mit Attributen, wo wir früher Eigenschaften verwendet haben.

Wir legen in Xcode über `Mac OS X > Application > Cocoa Application` ein neues Projekt an, aktivieren diesmal aber die Checkbox „Use Core Data for storage".

Nennen Sie das Projekt *CDBookshelf*. Unter *Groups & Files* sollten Sie eine neue Gruppe namens `Models` sehen. Sie enthält eine einzige Datei mit dem Namen `CDBookshelf_DataModel.xcdatamodel`. Öffnen Sie die Datei mit einem Doppelklick, und sobald Sie die erste Entität eingetragen haben, sollte das Ganze etwa so aussehen:



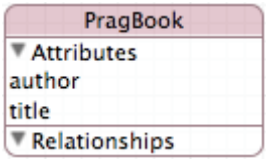
Am oberen Rand sehen Sie Subviews für Entitäten und Eigenschaften. Ich habe bereits eine Entität eingefügt, und das sollten Sie auch tun. Klicken Sie das Plus unten links im Entitätenbereich an. Geben Sie *PragBook* in das Namensfeld ein.[1] Der untere Teil des Fensters enthält eine grafische Darstellung Ihres Datenmodells. Sie erkennen `PragBook` ohne Attribute und Relationen.

Nun wollen wir Attribute für den Buchtitel und den Autorennamen einfügen. Wählen Sie `PragBook` im Entity-View und klicken Sie dann den Plus-Button am unteren Rand des Property-Views an. Es erscheint eine Pop-up-Liste mit den Optionen *Add Attribute*, *Add Fetched Property*, *Add Relationship* und *Add Fetch Request*.

Wählen Sie *Add Attribute* oder den Menüpunkt `Design > Data Model > Add Attribute` oder das Tastaturkürzel ⇧ ⌘ A. Nennen Sie das Attribut `title` und konfigurieren Sie es als nicht optional und vom Typ `String`. Da das Attribut nicht optional ist, sollten Sie ihm einen Standardwert wie „Book's title" zuweisen. Fügen Sie ein weiteres Attribut namens `author` ein, das ebenfalls nicht optional und vom Typ `String` ist.

1 Sie müssen das Namensfeld für *PragBook* verlassen, damit es auch im Entity-View erscheint.

Ihr Datenmodell sieht nun wie folgt aus:



Für's Erste ist unser Datenmodell fertig. Den Rest unserer Arbeit erledigen wir in der *.nib*-Datei.

22.2 Das Core Data-Widget nutzen

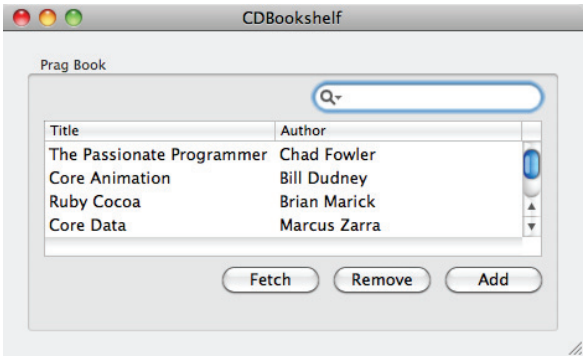
Legen Sie eine Kopie des Projekts an, da wir die Anwendung auf zwei unterschiedliche Arten abschließen wollen. Zuerst werden wir das von Apple bereitgestellte Widget nutzen, um gleichzeitig den Controller und den View zu erzeugen.

Klicken Sie `MainMenu.xib` doppelt an, um das Nib im IB zu öffnen. Ziehen Sie eine `Core Data Entity` aus der Library in Ihr `Window`-Objekt. Ein Assistent lässt Sie die `PragBook`-Entität aussuchen. Dazu wählen Sie zuerst das aktuelle Xcode-Projekt und dann das darin enthaltene Datenmodell. Projekte können mehr als ein Datenmodell umfassen. In unserem Fall gibt es aber nur eins. Innerhalb des Datenmodells gibt es üblicherweise mehrere Entitäten.

Unser Projekt enthält nur die `PragBook`-Entität. Wählen Sie sie aus und klicken Sie auf den *Next*-Button.

Auf der nächsten Seite können Sie sich für einen von drei Views entscheiden – ich wähle den Master/Detail-View. Ich habe auch die „Add/Remove"-Checkbox aktiviert, damit der Benutzer zur Laufzeit Bucheinträge einfügen und löschen kann. Zusätzlich habe ich die Suchfeld-Option aktiviert, damit der Benutzer die ausgegebenen Bücher nach bestimmten Suchkriterien filtern kann. Die „Detail Fields"-Checkbox wird nicht aktiviert. In unserem Beispiel erhalten wir keine zusätzlichen Informationen, wenn wir sie aktivieren, nur genau die Informationen, die auch in der Tabelle enthalten sind. Klicken Sie auf *Next*.

Auf der letzten Seite können Sie die Eigenschaften wählen, die in diesem Interface enthalten sind. Ich gebe sowohl `author` als auch `title` aus.



Klicken Sie auf *Build & Run*, und Sie verfügen über eine funktionierende Anwendung, die es Ihnen erlaubt, Bücher einzufügen und zu löschen und nach Einträgen zu suchen.[2]

Geben Sie einige Bücher ein und beenden Sie die Anwendung. Starten Sie sie erneut, und Sie werden feststellen, dass sie immer noch vorhanden sind. Vergewissern Sie sich, dass Sie ein oder mehrere Bücher löschen können und dass das Filtern der Liste durch Eingaben im Suchfeld möglich ist: Sie besitzen eine voll funktionsfähige Core Data-App, ohne eigentlich zu wissen, was da genau vor sich geht.

Nun wollen wir dieses Instant-Bücherregal beiseite legen und vom Datenmodell ausgehend den Controller und den View von Hand einbinden. Das hilft Ihnen dabei, die verschiedenen Schichten zu verstehen, die bei einer Core Data-Anwendung zum Tragen kommen.

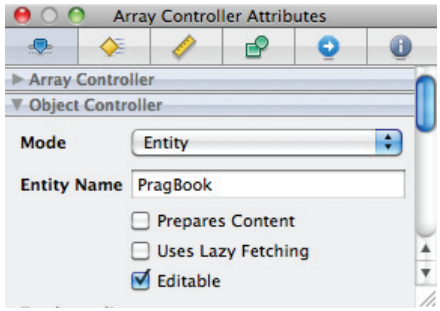
22.3 Der Managed Object-Kontext

Wir wollen nun zu unserer frischen Kopie des Projekts wechseln und noch einmal ein Datenmodell mit einer einzelnen Entität namens `PragBook` anlegen, die zwei Stringattribute namens `title` und `author` enthält. Öffnen Sie die *.nib*-Datei.

Fügen Sie einen Array-Controller in das Dokumentenfenster ein. Verwenden Sie den Identity Inspector, um den Namen des Array-Controllers auf `BookController` festzulegen. Beachten Sie, dass wir zwar den Namen, nicht aber die Klasse ändern. Wir müssen nur zwei kleine Änderungen vornehmen, um den Array-Controller einzurichten.

Zuerst öffnen Sie den Attributes Inspector für den `BookController`. Diesmal ist der Modus `Entity`, und der Name der Entität lautet *PragBook*.

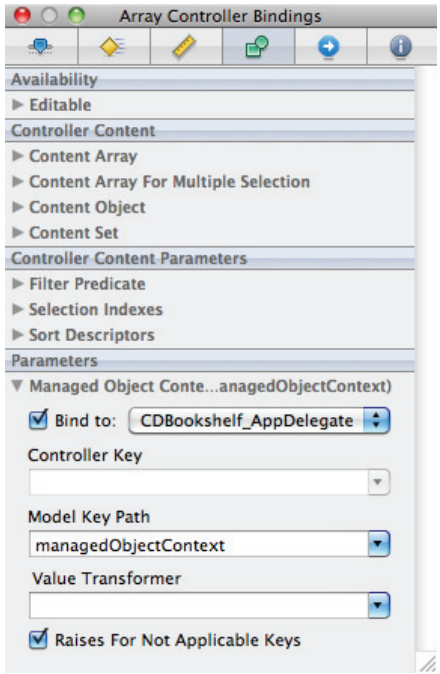
2 Siehe `CoreData/InstantBookshelf` im Codedownload.



Als Nächstes öffnen Sie den Bindings Inspector für den Array-Controller. Am unteren Ende des „Parameters"-Bereichs öffnen Sie das Dreieck für den `Managed Object Context`. Ich sage Ihnen, was Sie angeben müssen, und erkläre Ihnen dann, was Sie getan haben.

Aktivieren Sie die „Bind to"-Checkbox und wählen Sie `CDBookshelf_AppDelegate` aus der Drop-down-Liste. Die Einträge für `Controller Key` und `Value Transformer` bleiben leer. Setzen Sie den Model Key Path auf `managedObjectContext`.

Das Ganze sieht nun so aus:

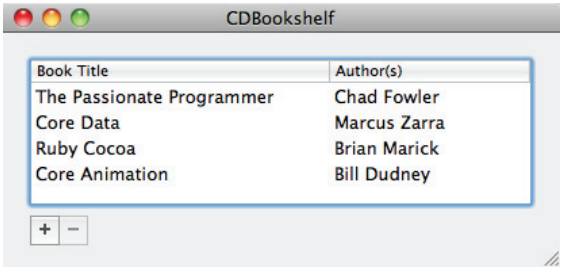


Der Managed Object-Kontext ist die Trennlinie zwischen Ihrer laufenden Anwendung und dem Mechanismus, der Daten auf der Platte speichert und von ihr abruft. Ihre laufende Anwendung interagiert mit dem Managed Object-Kontext über Objekte (genauer: „gemanagte“ Objekte), die zur Laufzeit aus Ihrem Datenmodell erzeugt werden.

Ein `NSManagedObject` oder eine Subklasse von `NSManagedObject` wird erzeugt, um jede Entität Ihres Datenmodells zu repräsentieren. Die eigentlichen Objekte werden dabei ganz nach Bedarf erzeugt.

Wir haben den Controller also in zwei Schritten mit dem Modell verknüpft. Der Controller musste mit dem Managed Object-Kontext durch den Application-Delegate verbunden werden. Sie mussten außerdem den Attributes Inspector benutzen, um festzulegen, für welche Entität dieser Controller verantwortlich ist.

Modell und Controller sind also gesetzt. Jetzt bauen wir den View genau so auf wie in Abschnitt 21.8, *Den View für unser Bücherregal entwickeln*, auf Seite 354. Das ist einer der Vorteile von MVC: Wir haben das Modell völlig geändert und müssen keinerlei Änderungen am View vornehmen.



Sie müssen nicht einmal die Verbindungen und Bindungen zwischen dem View und dem Controller ändern. Wie zuvor binden Sie die Tabellenspalten an den Controller, wobei Sie den Controller Key `arranged-Objects` und den Model Key Path `title` für die erste und `author` für die zweite Spalte verwenden. Die Buttons müssen Sie mit den `add:`- und `remove:`-Methoden des Controllers verbinden.

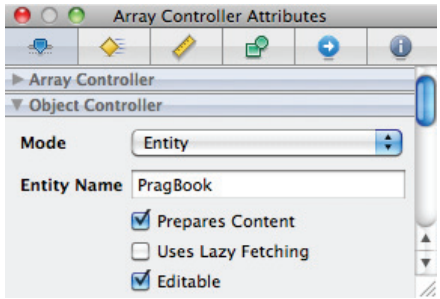
22.4 Die Persistenzschicht

Klicken Sie *Build & Run* an. Fügen Sie einige Bücher in Ihre Anwendung ein und beenden Sie sie dann. Starten Sie sie erneut. Moment mal! Was ist mit Ihren Daten passiert? Warum erwartet Sie eine leere Tabelle?

Denken Sie an Seite 273 in Kapitel 17, *Daten auf Festplatte speichern*, wo steht, dass Anwendungsdaten in Ihrem Homeverzeichnis unter `~/Library/Application Support` gespeichert werden. Das Verzeichnis hat den gleichen Namen wie die Anwendung, also sehen Sie in `CDBookshelf` nach und finden Sie die Datei `storedata`. Öffnen Sie `storedata` mit Ihrem Lieblingseditor, und Sie werden sehen, dass Ihre Daten gespeichert wurden und in etwa so aussehen:

```
<object type="PRAGBOOK" id="z106">
    <attribute name="title" type="string">
                              The Passionate Programmer</attribute>
    <attribute name="author" type="string">Chad Fowler</attribute>
</object>
```

Die Daten wurde also ohne Ihr Zutun auf der Festplatte gespeichert. Sie müssen eine einfache Änderung vornehmen, damit sie beim Start der Anwendung wieder eingelesen werden. Im Attributes Inspector von `Book Controller` aktivieren Sie die „Prepares Content“-Checkbox.



Klicken Sie *Build & Run* an, und Ihre Anwendung startet mit den Tabellenwerten, die Sie vorher eingegeben haben.[3] Um all das kümmert sich im Hintergrund der „Persistent Store Coordinator". Der Managed Object-Kontext interagiert mit der laufenden Anwendung, um basierend auf Ihrem Datenmodell die benötigten Objekte zu erzeugen. Er interagiert außerdem mit dem Persistent Store Coordinator, um Daten zu speichern und wieder abzurufen.

Wenn Sie sich die Codevorlage ansehen, die für Sie erzeugt wurde, dann erkennen Sie, dass in Ihrem Application-Delegate ein `NSPersistentStoreCoordinator` für Sie erzeugt wurde. Hier ein Fragment aus der `persistentStoreCoordinator`-Methode:

3 Das hier ist das Projekt `CDBookshelf1` im Codedownload.

CoreData/CDBookshelf1/CDBookshelf_AppDelegate.m

```
NSURL *url = [NSURL fileURLWithPath: [applicationSupportDirectory
►                    stringByAppendingPathComponent: @"storedata"]];
persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                               initWithManagedObjectModel: mom];
if (![persistentStoreCoordinator
►       addPersistentStoreWithType:NSXMLStoreType
        configuration:nil URL:url options:nil error:&error]){
```

In der ersten hervorgehobenen Zeile vervollständigen wir den Pfad für unseren Datenspeicher, indem wir `storedata` an den Pfad für den `CDBookshelf`-Ordner im `Application Support`-Verzeichnis anhängen. In der zweiten hervorgehobenen Zeile fügen wir den persistenten Speicher in den Persistent Store Coordinator ein und setzen seinen Typ auf `NSXMLStoreType`. Wir haben also festgelegt, das die Daten in XML gespeichert werden sollen, und die Datei benannt, in der diese Daten stehen sollen.

Sie werden während der Entwicklung häufig XML als Persistenzformat nutzen und vor dem Produktiveinsatz entscheiden, ob Sie zu SQLite oder einem eigenen Format wechseln. Die Arbeit mit XML erlaubt Ihnen, die Datei mit den gespeicherten Daten zu öffnen und sich anzusehen, was auf die Festplatte geschrieben wurde.[4]

Wenn es daran geht, Ihre Anwendung auszuliefern, können Sie bei Bedarf den Typ mit `NSSQLiteStoreType` angeben und den Namen der Datei ändern, in der die Daten gespeichert werden.

Bis auf Weiteres wollen wir weiterhin mit XML arbeiten.

22.5 Relationen

Wir wollen unser Datenmodell ein wenig verändern. Ein Buch kann mehr als einen Autor haben, und ein Autor könnte eine eigene Entität mit eigenen Attributen sein. Wir wollen das Ganze einfach halten, der Autor besteht also nur aus dem ersten und zweiten Vornamen sowie dem Nachnamen.

Da wir unser Datenmodell jetzt etwas komplexer gestalten wollen, sollten Sie das gesamte `CDBookshelf`-Verzeichnis aus dem `Application Support`-Ordner löschen.[5]

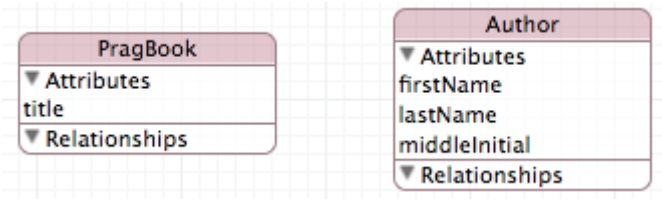
4 Klar,, wenn Sie gerne SQLite einsetzen, sind Sie wahrscheinlich damit vertraut, Befehle im Terminal einzugeben, um sich Ihre Tabellen anzusehen. Ich halte es aber für einfacher, mit XML zu beginnen und dann auf SQLite umzusteigen.

5 Es gibt Wege, um von einer Version des Schemas zum anderen zu migrieren, aber das würde jetzt zu weit führen.

Öffnen Sie nun das Datenmodell und wählen Sie das Attribut `author` in der Entität `PragBook`. Löschen Sie es über das Menüelement `Edit > Delete` oder durch Drücken der *Delete*-Taste.

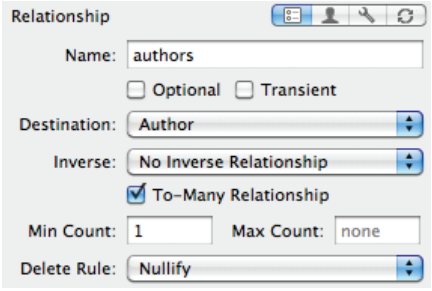
Legen Sie eine neue Entität namens `Author` an und weisen Sie ihr drei Attribute zu: `rstName`, `middleInitial` und `lastName`. Legen Sie sie als Strings an und stellen Sie sicher, dass der zweite Vorname optional ist. Setzen Sie die minimale und maximale Länge des zweiten Vornamen auf eins.

Das gibt uns zwei Entitäten, die mit nichts zu verbinden sind. Wir haben Bücher, und wir haben Autoren.



Wir wollen eine Relation (Beziehung) in `PragBook` herstellen, die auf die `Author`-Entität verweist. Jedes `PragBook` wird ein oder mehrere `Author`(s) besitzen. Wählen Sie `PragBook` im Properties-Bereich, klicken Sie das Plus an und fügen Sie eine Beziehung hinzu. Sie können auch das Menüelement `Design >DataModel >Add Relationship` oder das Tastaturkürzel CDR verwenden.

Nennen Sie die Beziehung `authors` und geben Sie die Entität `Author` als Ziel an. Es handelt sich um eine Mehrfachbeziehung („to-many relationship“), die nicht optional ist und bei der die Mindestanzahl eins lauten muss. Jedes Buch hat zumindest einen Autor, doch es kann auch mehrere geben.



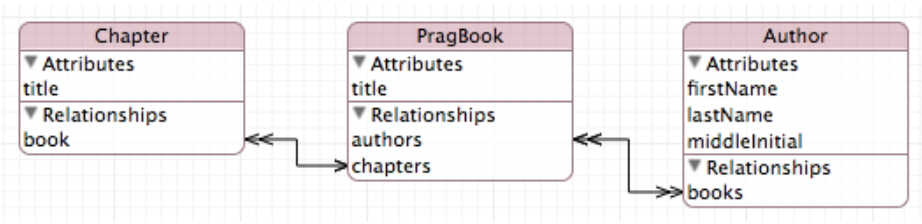
Ebenso kann jeder Autor mehr als ein Buch geschrieben oder zumindest daran mitgewirkt haben. Wir benötigen also auch eine Beziehung in der anderen Richtung. Wählen Sie `Author` und fügen Sie eine optionale To-many-Beziehung namens `books` mit dem Ziel `PragBook` ein. Diesmal klicken Sie das *Inverse Relationship*-Pull-down an und wählen `authors`.

Sie verfügen nun über zwei inverse Beziehungen zwischen Ihren Entitäten. Wenn Sie mit Core Data arbeiten, brauchen Sie immer eine Zwei-Wege-Relation, auch wenn Sie nicht glauben, dass Sie Verbindungen in beide Richtungen herstellen müssen.

Ein Buch besteht aus Kapiteln, weshalb wir die Entität `Chapter` einfügen wollen, die ein notwendiges Attribut `title` vom Typ String besitzt. Natürlich wäre ein reales Modell wesentlich komplexer, aber das würde uns nur dabei stören, die Grundlagen der Arbeit mit Core Data zu erläutern.

Da ein Buch ein oder mehrere Kapitel enthält, wählen Sie `PragBook` und legen eine neue Relation namens `chapters` an. Es handelt sich um eine erforderliche Beziehung mit der Zielentität `Chapter`. Wählen Sie `Chapter` und legen Sie eine inverse Beziehung namens `book` an. Auch wenn manche Verlage einzelne Kapitel für andere Zwecke weiterverwenden, gehört in unserem Fall jedes Kapitel zu nur einem Buch, es handelt sich also nicht um eine To-many-Beziehung. Sie ist allerdings nicht optional und besitzt die Zielentität `PragBook` und die inverse Beziehung `chapters`.

Unser Entity-Relationship-Diagramm sieht nun so aus:



Wir besitzen ein einfaches Datenmodell mit drei Entitäten, einer Hand voll Attribute und einer Relation zwischen allen Entitätenpaaren. Ein weiterer Schritt ist noch nötig, um unser Modell fertigzustellen.

22.6 Die Löschregel einer Relation wählen

Bevor wir das Datenmodell abschließen, müssen wir uns überlegen, was passiert, wenn der Benutzer Daten löscht. Wenn er zum Beispiel ein Buch löscht, sollten auch die Kapitel des Buches gelöscht werden. Aber wenn der Benutzer ein Kapitel löscht, soll ja nicht das ganze Buch gelöscht werden. Und wenn der Benutzer ein Buch löscht, was soll dann mit den Autoren passieren?

Wir haben verschiedene Optionen, wenn der Benutzer einen Eintrag löscht: *Deny* (ablehnen), *Nullify* ("ausnullen"), *Cascade* (kaskadieren) und *No Action* (keine Aktion). Die Löschregeln sind Teil der Definition einer Relation. Sehen wir uns einige Beispiele in unserem aktuellen Modell an.

Fangen wir mit der Löschregel für die `chapters`-Relation an. Wie sollte sie aussehen? Wir wollen sicherstellen, dass, wenn in einem Buch noch Kapitel enthalten sind, der Benutzer das Buch nicht löschen kann, das diese Kapitel enthält.

Mit anderen Worten wählen wir die *Deny*-Option, wenn der Benutzer ein Buch löschen will, für das noch Kapitel existieren (die ohne das Buch verwaisen würden). Die *Deny*-Option verhindert das Löschen der Entität, wenn das Ziel der Relation noch über mindestens ein Element verfügt.

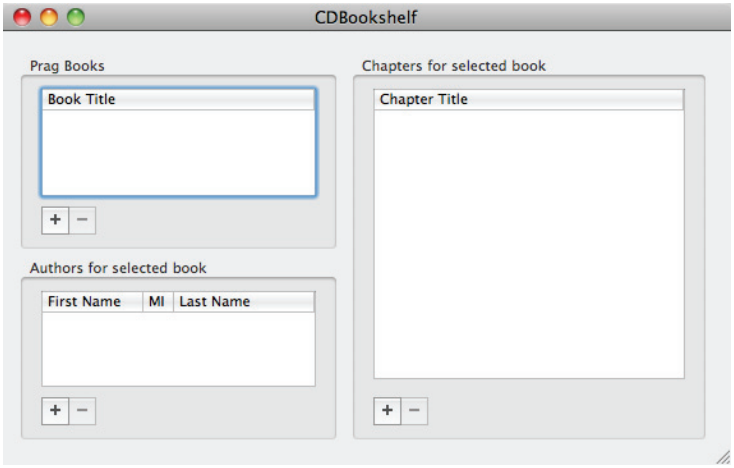
Bei Autoren ist es anders. Ein Autor kann mehr als ein Buch geschrieben haben; wir wollen also die Autoren nicht löschen, wenn wir ein Buch löschen. In diesem Fall wählen wir die Option *Nullify*. Mit *Nullify* unterbinden wird das Löschen des Buches nicht, wenn ein oder mehrere Autoren noch existieren. Wir setzen die `book`-Relation für jeden Autor auf null. Wir nullen die Relation aus, weil das Buch nicht länger existiert und der Autor es somit nicht geschrieben haben kann.

Für die `chapters`-Relation gibt es noch eine weitere Option: Wenn wir ein Buch löschen, können wir gleichzeitig auch alle Kapitel des Buches löschen. Mit anderen Worten wollen wir die Löschung kaskadieren. Wir verfolgen dabei immer noch den Ansatz, dass ein Kapitel nicht überleben kann, wenn das Buch nicht existiert – wir haben es nur mit einem anderen Ergebnis zu tun.

Also legen wir die Löschregel für `chapters` mit *Cascade* fest. Die inverse Relation `book` muss die Löschregel *Nullify* verwenden. Wenn wir ein Kapitel löschen, löschen wir nicht das gesamte Buch, sondern stellen sicher, dass das Buch nicht mehr auf das gelöschte Kapitel verweist. Dementsprechend muss die Regel für `authors` und `books` *Nullify* sein.[6]

22.7 Den View aktualisieren

An diesem Punkt sollten Sie mit dem Interface Builder so vertraut sein, dass Sie keine großen Anleitungen mehr benötigen. Öffnen Sie die *.nib*-Datei, wählen Sie den Tabellen-View und reduzieren Sie die Anzahl der Spalten auf 1. Wählen Sie alle Komponenten innerhalb des Content-Views aus und wählen Sie dann `Layout > Embed Objects In > Box`. Ändern Sie den Titel der Box in *Prag Books*. Kopieren Sie diese Box, fügen Sie sie zweimal ein und passen Sie ihren Inhalt so an, dass er wie folgt aussieht:



Der Inhalt der Kapitel- und Autoren-Boxen hängt vom gewählten Buch ab. Das ist Aufgabe der Controller-Schicht.

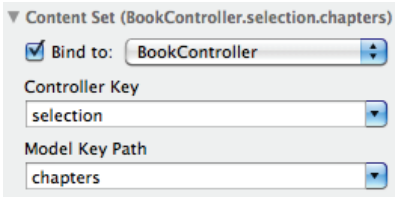
22.8 Abhängigkeiten verwalten

Damit Benutzer Kapitel in ein Buch einfügen können, folgen wir den gleichen Schritten wie beim `Book Controller`. Ziehen Sie einen Array-Controller aus der Library in das Dokumentenfenster. Im Identity Inspector ändern Sie seinen Namen in `Chapter Controller`. Im Attribu-

6 Sie können Ihre Arbeit mit dem Datenmodell aus `CDBookshelf2` im Codedownload vergleichen.

tes Inspector ändern Sie den Modus in *Entity* und den Entitätsnamen in `Chapter`. Diesmal aktivieren Sie „Prepares Content“ nicht. Im Bindings Inspector binden Sie den Managed Object Context an den `CDBookshelf_AppDelegate` und setzen den Model Key Path auf `managedObjectContext`.

Nun folgt etwas ganz Neues. Jedes Buch besitzt einen eigenen Satz Kapitel. Der `Chapter Controller` verwaltet die Kapitel des gerade gewählten Buches. Wie Sie gleich sehen werden, müssen die Kapitel wechseln, sobald das Buch gewechselt wird. Im Bindings Inspector müssen Sie das Content Set für den `Chapter Controller` mit dem richtigen Schlüsselpfad im `Book Controller` verknüpfen. Der Controller Key muss daher `selection` und der Model Key Path `chapters` lauten.



Bleibt noch die Verknüpfung des Views. Wenn Sie die Kapiteltabelle und die Buttons angelegt haben, indem Sie die Buchtabelle kopiert haben, sind die Elemente wahrscheinlich mit dem `BookController` verknüpft. Verschieben Sie alles in den `ChapterController`.

Wählen Sie den Plus-Button im Chapter-Abschnitt. Im Connections Inspector verbinden Sie den `selector:` mit dem `Chapter Controller`. Wenn das Pop-up mit den verfügbaren Verbindungen erscheint, wählen Sie `add:`. In gleicher Weise verbinden Sie den Minus-Button-`selector:` mit der `Chapter Controller-remove:`-Methode.

Wir müssen auch die Bindungen der Buttons festlegen. Wählen Sie erneut den Plus-Button und verwenden Sie den Bindings Inspector, um *Enabled* an den `Chapter Controller` mit dem Controller Key `canAdd` zu binden. In gleicher Weise binden Sie *Enabled* für den Minus-Button mit dem Controller Key `canRemove`.

Zum Schluss müssen Sie die Tabellenspalte an den `Chapter Controller` binden. Wie zuvor binden Sie *Value* mit der Controller Key-Einstellung `arrangedObjects`, weil wir eine Spalte an den Inhalt eines Arrays binden. Sie verwenden die Model Key Path-Einstellung `title`. Ich mache eine kurze Pause, während Sie diese Schritte in der folgenden Übung für die Autorentabelle noch einmal nachvollziehen.

22.9 Übung: Autoren hinzufügen und löschen

Fügen Sie einen `Author Controller` in ihre *.nib*-Datei ein. Er muss mit dem Managed Object-Kontext verbunden sein und mit den `Author`-Entitäten arbeiten, die mit dem gewählten Buch verknüpft sind. Verknüpfen und binden Sie die visuellen Elemente von `Author Controller` in der gleichen Weise, wie Sie es für den `Chapter Controller` getan haben.[7]

Klicken Sie auf *Build & Run*: Ohne irgendwelchen Programmcode schreiben zu müssen, besitzen Sie eine voll funktionsfähige Anwendung, die es Ihnen erlaubt, Buchtitel und dazugehörige Kapitelnamen einzufügen und zu löschen. Ihre Buttons werden korrekt aktiviert bzw. deaktiviert, und die Informationen werden auf der Festplatte gespeichert und wieder eingelesen. Ein „Rückgängigmachen" gibt es gratis dazu. Fügen Sie einige Bücher mit entsprechenden Kapiteln ein. Wählen Sie nun `Edit > Undo` oder ⌘Z, und Sie können Ihre Arbeit Schritt für Schritt rückgängig machen.

22.10 Sortieren

Ihnen könnte ein kleines Problem aufgefallen sein: Wenn Sie die Anwendung beenden und neu starten, einen Kapiteleintrag rückgängig machen oder einfach von Buch zu Buch wechseln, ändert sich die Reihenfolge der Kapitel. Der Grund dafür ist, dass es sich bei den Inhalten um Mengen handelt, dass es also keine feste Reihenfolge gibt. Sie können die Kapitel eines Buches in der richtigen Reihenfolge eingeben, aber es gibt keine Garantie dafür, dass diese Reihenfolge auch beim nächsten Start oder beim nächsten Wechsel zu diesem Buch erhalten bleibt. Über die Jahre gab es darüber in der Apple-Liste *cocoa-dev* einige Diskussionen.[8] Leute wie Tim Isted[9] und Brian Webster[10] haben nützliche Workarounds entwickelt, die weit über den Rahmen dieses Buches hinausgehen.

---

7 Sie finden die Lösung in `CDBookshelf3` im Codedownload.

8 Sehen Sie sich auch diesen Thread an:
http://www.cocoabuilder.com/archive/message/cocoa/2005/ 6/14/138793.

9 http://www.timisted.net/blog/archive/core-data-drag-drop/

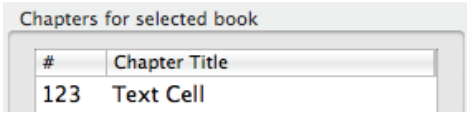
10 http://www.fatcatsoftware.com/blog/2008/per-object-ordered-relationships-using-core-data

Wir wollen einen einfacheren Ansatz nutzen und ein weiteres Attribut einfügen, das die Kapitelnummern speichert und uns die Sortierung der Kapitel basierend auf dieser Liste erlaubt. Bevor wir beginnen, löschen wir das `CDBookshelf`-Verzeichnis aus `Application Support`.

Öffnen Sie Ihr Datenmodell in Xcode. Wählen Sie die Entität `Chapter` und fügen Sie ein neues Attribut namens `chapterNumber` vom Typ `Integer 16` ein. Speichern Sie ab und wechseln Sie in Ihre *.nib*-Datei.

Im Interface Builder fügen Sie eine zweite Spalte in die Tabelle im Chapters-Bereich ein. Ordnen Sie die beiden Spalten so an, dass die neue Spalte links steht und recht schmal ist. Als Titel tragen Sie einfach `#` ein. Binden Sie den Spaltenwert an den `Chapter Controller`. Setzen Sie diesen auf `arrangedObjects` und den Model Key Path auf `chapterNumber`.

Wir müssen eine kleine Korrektur vornehmen: Die Spalte erwartet eine Eingabe als `NSNumber`, doch wenn Sie etwas in das Feld eintragen, wird es als `NSString` eingelesen. Um das zu beheben, wählen Sie einen Zahlenformatierer aus der Library und ziehen ihn in die Tabellenzelle. Im Attributes Inspector legen Sie ein Dezimalformat mit null Nachkommastellen fest. Der Formatierer muss Mac OS X 10.4+ Custom unterstützen. Sie wissen, dass Sie es richtig eingetragen haben, wenn „123" anstelle von „Text Cell" erscheint.



Nun wollen wir beide Spalten so einrichten, dass sie nach der Kapitelnummer sortiert werden. Wählen Sie die erste Spalte und öffnen Sie den Attributes Inspector. Legen Sie für Sort Key `chapterNumber` fest und für Selector `compare:`. Tun Sie das Gleiche mit der zweiten Spalte – und stellen Sie sicher, dass immer noch nach `chapterNumber` sortiert wird.

Klicken Sie *Build & Run* an. Geben Sie die Kapitelnummern und -namen für ein Buch ein. Geraten die Kapitel durcheinander, können Sie die Tabelle neu anordnen, indem Sie die Überschrift der jeweiligen Spalte anklicken. Ich bin noch nicht ganz zufrieden mit diesem Verhalten. Ich möchte die Anwendung starten, und die Kapitel sollen standardmäßig sortiert sein. Leider ist dazu ein wenig Programmcode notwendig, weshalb das noch warten muss.

Bevor wir weitermachen, möchte ich, dass Sie eine Pause einlegen und sich ansehen, wie weit wir in diesem Kapitel gekommen sind, ohne irgendwelche Code zu entwickeln. Wir besitzen eine Anwendung, die Daten auf der Festplatte speichert und es dem Benutzer ermöglicht, Informationen zu Büchern, Autoren und Kapiteln anzulegen und zu modifizieren.[11] Als Nächstes wollen wir den Benutzer die Buchliste filtern lassen.

22.11 Elemente filtern

Sobald Ihre Buchliste anwächst, werden Sie sie filtern wollen, um sie basierend auf irgendeinem Kriterium einzugrenzen. In einem ersten Schritt wollen wir nach dem Buchnamen filtern. Ich wünsche mir ein Suchfeld, in das ich einen Teil des Titels eingeben kann. Sobald ich etwas eingebe, soll die Liste der dargestellten Buchtitel auf die Bücher eingeschränkt werden, die meine Eingabe enthalten. Das soll so funktionieren wie das Suchfeld am unteren Rand des Library-Fensters im Interface Builder.

Wir sind in der Lage, die ganze Arbeit im Nib zu erledigen. Suchen Sie sich ein Suchfeld aus der Library heraus und platzieren Sie es innerhalb des Fensters. Sie können auch ein Label einfügen, um den Benutzer wissen zu lassen, wonach er suchen kann.

Wählen Sie das Suchfeld aus und benutzen Sie den Bindings Inspector, um die Suchkriterien festzulegen. In der *Search*-Gruppe öffnen Sie das *Predicate*-Dreieck. Wir binden diese Suche an den `Book Controller` mit einem Controller- Schlüssel von `filterPredicate`.

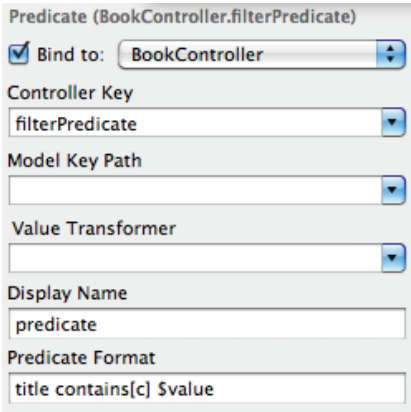
Die Suche erfolgt durch den Aufbau und die Übergabe eines Prädikats. Prädikate können recht kompliziert sein und mit Programmcode konstruiert werden, wenn sie zu unübersichtlich dafür werden, dass man sie bequem in den vorhandenen Platz eintragen kann.[12] In unserem Fall suchen wir nach Büchern, deren Titel den vom Benutzer eingegebenen Text enthalten. Wir sorgen außerdem dafür, dass bei der Suche nicht zwischen Groß- und Kleinschreibung unterschieden wird:

```
title contains[c] $value
```

11 Vergleichen Sie Ihre Version mit `CDBookshelf4` im Codedownload.

12 Siehe Apples *Predicate Programming Guide*.

Kurz gesagt, haben Sie den Bindings Inspector des Suchfelds so modifiziert:



Wenn ich jetzt *Core* in das Suchfeld eintrage, wird meine Liste aller Pragmatic Bookshelf-Bücher auf *Core Animation* und *Core Data* reduziert. Vergleichen Sie Ihre Ergebnisse mit `CDBookshelf5`.

Diese Suche ist weder interessant noch besonders nützlich. Ich hätte die Liste der Buchtitel, die *Core* enthalten, auch selbst bestimmen können. Doch was ist, wenn ich die Hierarchie auf den Kopf stellen will? Lassen Sie uns nach allen Büchern suchen, die von einem bestimmten Autor geschrieben wurden. Das von Hand zu machen, wäre eine Qual. Wir müssten jedes Buch wählen, uns die Autorennamen ansehen und prüfen, ob sie unserem Kriterium entsprechen. Die Automatisierung dieser Suche ist eine gute Sache und nicht besonders aufwendig.

Wir binden unsere Suche immer noch an den `Book Controller`. Diesmal durchsuchen wir alle Autoren aller Bücher und vergleichen ihre Nachnamen mit dem vom Benutzer eingegebenen String. Dazu ersetzen wir das „Predicate Format“ wie folgt:

```
any authors.lastName contains[c] $value
```

Wenn Sie nun *Dudney* eingeben, gibt meine Buchliste das von Bill geschriebene *Core Animation*- und das von ihm mitgeschriebene *iPhone SDK*-Buch aus. Ganz schön leistungsfähig. Andernfalls hätten Sie nacheinander jedes Buch wählen und sich die Liste der Autoren ansehen müssen.[13]

13 Diese Version der Suche ist `CDBookshelf6` im Codedownload.

22.12 Den Sortierdeskriptor programmieren

Alles, was Sie im Interface Builder erledigen können, lässt sich auch mit Programmcode machen. Tatsächlich sind einige der Dinge, die wir mit dem Interface Builder erledigt haben, nur sehr dünne Schichten über den Methodenaufrufen. Als Sie beispielsweise die Sortierung für Tabellenspalten eingefügt haben, haben Sie den Methodennamen angegeben, als Sie `compare:` als Selektor festlegten.

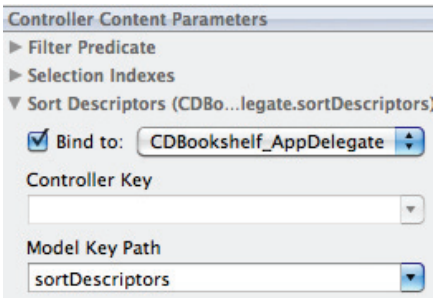
Denken Sie daran, dass wir noch nicht über das gewünschte Verhalten verfügen. Die Kapiteltabelle soll sortiert sein, sobald sie erscheint. Dazu benötigen wir ein wenig Code. Deklarieren Sie eine Instanzvariable namens `sortDescriptors` vom Typ `NSArray` und eine dazugehörige Eigenschaft mit dem gleichen Namen in der `CDBookShelf_AppDelegate`-Header-Datei.

Synthetisieren Sie den Sortierdeskriptor und initialisieren Sie ihn in der `applicationDidFinishLaunching:`-Methode des Delegates so, dass er das `chapterNumber`-Attribut als Schlüssel verwendet:

CoreData/CDBookshelf7/CDBookshelf_AppDelegate.m

```
@synthesize window, sortDescriptors;

-(void) applicationDidFinishLaunching: (NSNotification *) notification {
   self.sortDescriptors = [NSArray arrayWithObject:
           [[NSSortDescriptor alloc] initWithKey:@"chapterNumber"
                                       ascending:YES]];
}
```

Nachdem wir die `sortDescriptors` erzeugt und initialisiert haben, können wir die Sortierdeskriptoren von `Chapter Controller` wie folgt binden:



Klicken Sie auf *Build & Run*: Jetzt bleiben die Kapitel vom Programmstart bis zum Programmende in der richtigen Reihenfolge.

In diesem Kapitel habe ich beschrieben, wie man mit Core Data arbeitet, wenn man Anwendungen für Mac OS X entwickelt. Denken Sie daran, dass das iPhone OS noch keine Bindungen kennt, weshalb die Einzelheiten bei der Arbeit mit Core Data da ein wenig anders aussehen. Sie definieren die Daten in der gleichen Weise, rufen sie aber etwas anders ab.

Es ist nicht möglich, Core Data in ein oder zwei Kapiteln komplett abzuhandeln. Marcus Zarras Buch *Core Data* [Zar09] zeigt einige der fortgeschritteneren Ideen. Sie sollten sich auch Apples *Introduction to Core Data Programming* [App09d] ansehen.

Ein Großteil des Codes, den Sie werden schreiben müssen, fügt sich gut in die Controller-Schicht oder den App-Delegate ein. Es gibt aber Zeiten, in denen Sie Änderungen oder Ergänzungen an einer Klasse vornehmen wollen, die eine Entität repräsentiert. Im nächsten Kapitel werden Sie eine Technik kennenlernen, die es einfacher macht, Änderungen an einer Klasse außerhalb dieser Klasse vorzunehmen. Diese mächtige Technik wird *Kategorien* genannt.

Kapitel 23

Kategorien

In Abschnitt 17.5, *Ein Archiv auf Festplatte speichern*, auf Seite 280 wollten wir ein `NSSet` auf Platte speichern und später wieder einlesen. Das wäre einfach gewesen, wenn wir mit einem `NSArray` gearbeitet hätten. Ein `NSArray` besitzt fest eingebaute Methoden, mit denen es sich selbst auf Platte speichern und ein Array aus einer Datei erzeugen kann. `NSSet` besitzt diese Fähigkeit nicht.

Kategorien (*Categories*) erlauben uns, diese Einschränkungen zu umgehen. Wir verwenden Kategorien, um neue Methoden für existierende Klassen zu deklarieren und zu implementieren. Dabei kann es sich um von uns entwickelte Klassen handeln oder um Klassen, bei denen wir keinen Zugriff auf den Quellcode haben.

In diesem Kapitel wollen wir Kategorien nutzen, um einem Set beizubringen, wie es sich selbst auf Platte speichern und wieder einlesen kann. Danach werden wir eine Variante von Kategorien nutzen, um private Methoden in unserem Code zu deklarieren. Zum Schluss werden wir Kategorien mit Core Data einsetzen, um das Verhalten von Objekten auf zwei verschiedene Arten zu modifizieren.

23.1 Beschränkungen überwinden

Wir beginnen mit einer einfachen Anwendung, die ein Array auf Festplatte schreibt und wieder einliest. Legen Sie eine neue Cocoa-Anwendung namens *Bounce* an. Stellen Sie sicher, dass die Checkbox „Use Core Data for Storage“ nicht aktiviert ist. Unsere Anwendung erzeugt ein Array, speichert es auf Platte, erzeugt aus dem, was auf der Platte gespeichert ist, ein zweites Array und gibt es über die Logfunktion aus.

Die einfachste Lösung besteht darin, den Code in die `applicationDidFinishLaunching:`-Methode in `BounceAppDelegate.m` zu packen:

Categories/Bounce1/BounceAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
   NSArray *source = [NSArray arrayWithObjects:@"One",@"Two", nil];
   [source writeToFile:@"savedArray" atomically:YES];
   NSArray *fromDisk = [NSArray arrayWithContentsOfFile:@"savedArray"];
   NSLog(@"Array from disk: %@",fromDisk);
}
```

Klicken Sie *Build & Run* an, und alles funktioniert wunderbar. Unser erstes Array wird erzeugt und auf der Festplatte gespeichert. Unser zweites Array wird aus dem gespeicherten Inhalt erzeugt und Folgendes in der Konsole ausgegeben:

```
Array from disk: (
    One,
    Two
)
```

Ich möchte das Gleiche mit `NSSet` anstelle von `NSArray` machen. Mit anderen Worten möchte ich die Implementierung der `applicationDidFinishLaunching:`-Methode wie folgt ändern, und es soll einfach funktionieren:

Categories/Bounce2/BounceAppDelegate.m

```
-(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
►  NSSet *source = [NSSet setWithObjects:@"One",@"Two", nil];
   [source writeToFile:@"savedSet" atomically:YES];
►  NSSet *fromDisk = [NSSet setWithContentsOfFile:@"savedSet"];
   NSLog(@"Set from disk: %@",fromDisk);
}
```

Der Compiler beschwert sich, dass `NSSet` möglicherweise nicht auf die Methoden `writeToFile:atomically:` und `setWithContentsOfFile:` reagiert. Und wenn wir diese Warnungen ignorieren und die Anwendung trotzdem ausführen, kommt es zu einem Laufzeitfehler, weil der Selektor `writeToFile:atomically:` nicht existiert.

Das wollen wir korrigieren. Ein `NSSet` weiß nicht, wie es sich selbst auf Festplatte speichern kann. Wir wollen `NSSet` diese Fähigkeit verleihen, indem wir es um eine Kategorie namens `Persistence` erweitern.

23.2 Eine Kategorie anlegen

Legen Sie eine neue Objective-C-Klasse an, die `NSObject` erweitert. Ich weiß, dass wir eigentlich eine Kategorie anlegen und keine Klasse, doch momentan gibt es für Kategorien noch keine Vorlage. Per Konvention besteht der Dateiname aus dem Namen der Kategorie und dem Namen der Klasse, für die sie eine Kategorie ist. Wir wollen unsere Kategorie `Persistence` nennen. Wir erweitern die Funktionalität der `NSSet`-Klasse, die Implementierungsdatei der Kategorie heißt also `NSSet+Persistence.m`.

Eine Kategorie kann keine Instanzvariablen enthalten. Da Sie die Klasse erweitern, nachdem der Speicher alloziert und die Variablen initialisiert sind, können Sie die Klasse um neue Methoden erweitern, aber das war es dann auch schon.

Das erinnert ein wenig an ein Protokoll, doch es gibt einen großen Unterschied. Wie Sie in Kapitel 12, *Protokolle für die Delegation entwickeln*, auf Seite 203 gesehen haben, ist ein Protokoll eine Sammlung von Methodendeklarationen, die optional oder vorgeschrieben sein können. Jeder Klasse steht es frei, ein Protokoll zu adaptieren. Eine Klasse, die in ihrer Header-Datei verspricht, sich an ein Protokoll zu halten, muss die entsprechenden Methoden auch irgendwie implementieren.

Eine Kategorie ist hingegen an eine bestimmte Klasse gebunden, die im Interface und der Implementierung angegeben werden muss. Die Kategorie muss außerdem die deklarierten Methoden implementieren. Die Methoden, die Sie in einer Kategorie deklarieren und implementieren, entsprechen denen, die Sie auch in der Header- und Implementierungsdatei einer Klasse deklarieren und implementieren.

Bauen Sie die Header-Datei so auf:

Categories/Bounce3/NSSet+Persistence.h

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

@interface NSSet(Persistence)

- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag;
+ (id)setWithContentsOfFile:(NSString *)aPath;

@end
```

Zu dieser kleinen Datei gibt es eine ganze Menge anzumerken. Sie können sehen, wie man festlegt, dass dies das Interface für die `Persistence`-Kategorie ist, die Sie für `NSSet` definieren. Wir lassen die geschweiften Klammern weg, da es keine Instanzvariablen geben kann.

Für die von uns deklarierten Methoden habe ich die Signatur der `writeToFile:atomically:`-Methode aus `NSArray` kopiert und die Signatur von `arrayWithContentsOfFile:` adaptiert.

Hier das Grundgerüst der zugehörigen Implementierungsdatei:

Categories/Bounce3/NSSet+Persistence.m

```
#import "NSSet+Persistence.h"

@implementation NSSet(Persistence)

@end
```

Wir müssen nur die beiden Methoden implementieren, die wir in der Header-Datei deklariert haben. Erinnern Sie sich daran, dass unsere Strategie darin besteht, unser Set in ein Array umzuwandeln (und umgekehrt) und dann die Array-Methoden zum Schreiben und Lesen zu verwenden. Glücklicherweise besitzt `NSSet` die Klassenmethode `setWithArray:`, die ein neues `NSSet` aus einem `NSArray` erzeugt. Wir können das wie folgt mit `NSArrays` `arrayWithContentsOfFile:` verknüpfen:

Categories/Bounce3/NSSet+Persistence.m

```
+ (id)setWithContentsOfFile:(NSString *)aPath {
    return [NSSet setWithArray:[NSArray arrayWithContentsOfFile:aPath]];
}
```

Auf Platte zu schreiben, ist etwas schwieriger, weil `NSArray` leider keinen passenden Konstruktor besitzt, um ein Array aus einem Set zu erzeugen. Erzeugen Sie ein `NSMutableArray` mit der Größe von `NSSet`. Gehen Sie dann alle Elemente des Sets durch und fügen Sie sie in das Array ein. Sobald Sie das getan haben, kann sich das Array selbst auf die Festplatte schreiben.

Categories/Bounce3/NSSet+Persistence.m

```
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag {
    NSMutableArray *temp = [NSMutableArray arrayWithCapacity:self.count];
   for (id element in self){
       [temp addObject:element];
   }
   return [temp writeToFile:path atomically:YES];
}
```

Beachten Sie, dass wir alle Methoden der erweiterten Klasse nutzen und auch auf alle Instanzvariablen zugreifen können. Wir verwenden `count` und iterieren über alle Elemente des zugrunde liegenden `NSSet`-Objekts.

Klicken Sie auf *Build & Run*. Der Compiler warnt Sie immer noch davor, dass `NSSet` möglicherweise nicht auf die Methoden `writeToFile:`

`atomically:` und `setWith-ContentsOfFile:` reagiert. Doch diesmal gibt es keinen Laufzeitfehler: Die Anwendung funktioniert ausgezeichnet. Das Set wird erzeugt, und die in der Kategorie vorhandenen Methoden kümmern sich um das Schreiben und Lesen der Daten.

23.3 Sicherheitshinweise zu Kategorien

Haben Sie bemerkt, was gerade passiert ist?

Wir haben etwas Clientcode entwickelt, der zwei Methoden aufgerufen hat, die in der Klasse `NSSet` nicht existierten. Der Compiler hat sich beschwert und es kam zu einem Laufzeitfehler. Wir haben das Problem gelöst, ohne diesen Code anzurühren und ohne Zugriff auf den `NSSet`-Quellcode zu haben.

Wir haben `NSSet` um eine Kategorie erweitert und die Methoden eingefügt, die wir in dieser Kategorie benötigten. Sicher, der Compiler hat sich immer noch beschwert, aber die Anwendung hat funktioniert.

Wie fühlen Sie sich dabei?

Ihre Antwort gibt wahrscheinlich mehr über Ihren Programmierhintergrund preis als alles andere. Rubyisten werden daran nichts Besonderes finden können – das sind einfach Mix-ins. Sie haben sich schon seit über einem Dutzend Kapiteln gefragt, wann sie endlich auftauchen. Wenn Sie aber mit einer statisch typisierten Sprache aufgewachsen sind, brummt Ihnen der Kopf: Woher sollen Sie wissen, worauf Sie sich verlassen können? Die Grundlagen, auf denen Sie Ihren Code aufbauen, können von anderen über Kategorien geändert werden.

Das stimmt, aber in der Praxis passiert es nicht. Verwenden Sie Kategorien nicht, um vorhandene Methoden zu verändern. Wenn Namen miteinander kollidieren könnten, sollten Sie dem Namen etwas voranstellen, das ihn entsprechend abhebt. Ich könnte beispielsweise `PP` für Pragmatic Programmers verwenden und die Methoden `PP_writeToFile:` und `PP_setWithContentsOfFile:` nennen.

Ich möchte es außerdem explizit dazusagen, wenn mein Code eine Kategorie *verwendet*. Fügen Sie die folgende Importanweisung am Anfang von `BounceAppDelegate.m` ein:

```
#import "NSSet+Persistence.h"
```

Sobald das geschehen ist, verschwindet auch die Compiler-Warnung. Die Klasse weiß nun um die zusätzlichen Methoden.

Wir werden Kategorien gleich auf unser Core Data-Beispiel anwenden, doch vorher wollen wir uns eine besondere Form von Kategorie ansehen: eine ohne Namen.

23.4 Private Methoden in Klassenerweiterungen

Nehmen wir an, wir wollen die `applicationDidFinishLaunching:`-Methode des `BounceAppDelegate`s refaktorieren. Im Moment ist alles ein wüstes Durcheinander.

Categories/Bounce3/BounceAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSSet *source = [NSSet setWithObjects:@"One",@"Two", nil];
    [source writeToFile:@"savedSet" atomically:YES];
    NSSet *fromDisk = [NSSet setWithContentsOfFile:@"savedSet"];
    NSLog(@"Set from disk: %@",fromDisk);
}
```

Joe fragt...

Warum kann man der Kategorie keinen Namen geben?

Man könnte. Nennen Sie sie `Private` und deklarieren Sie sie so:

```
@interface BounceAppDelegate(Private)
```

... und zwar zu Beginn von `BounceAppDelegate.m`. Das ist ein anderes Konstrukt, das als *informelles Protokoll* bezeichnet wird. Um ein informelles Protokoll zu erzeugen, deklarieren Sie das Interface für eine benannte Kategorie ohne die dazugehörige Implementierung. Die Klasse und jede Subklasse der Klasse, mit der die Kategorie verknüpft ist, können die darin deklarierten Methoden implementieren oder auch nicht.

Ich verschwende nicht viel Zeit auf informelle Protokolle, da Sie so gut wie nie welche deklarieren werden. Da Methoden in Protokollen mittlerweile als optional deklariert werden können, ersetzt Apple tatsächlich einige der informellen Protokolle durch formelle. Sehen Sie sich die Beschreibung von `numberOfRowsInTableView:` in der Protokollreferenz zu `NSTableViewDataSource` an. Unter „Availability" sehen Sie, dass diese Methode vor Snow Leopard noch Teil eines informellen Protokolls war. Nun ist sie Teil eines formellen – wie ich es in diesem Buch einfach als *Protokoll* bezeichnet habe.

Lassen Sie uns eine Methode namens `createSetOnDisk` mit den ersten beiden Zeilen und eine namens `setFromDisk` mit den letzten beiden entwickeln. Hier die refaktorierte `applicationDidFinishLaunching:`-Methode:

Categories/Bounce4/BounceAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
   [self createSetOnDisk];
   [self setFromDisk];
}
```

Wo würde man die neuen Methoden `createSetOnDisk` und `setFromDisk` unterbringen?

Bisher haben wir nur zwei Möglichkeiten. Eine wäre, die Methoden im Quellcode vor `applicationDidFinishLaunching:` unterzubringen, damit sich der Compiler nicht beschwert, dass die `BounceAppDelegate`-Klasse diese Methoden möglicherweise nicht implementiert. Die andere Möglichkeit wäre, die Methoden in der Header-Datei zu deklarieren. Dann kann man sie unterbringen, wo man will.

In diesem kleinen Beispiel spielt das eigentlich keine Rolle, aber es gibt Situationen, in denen mir beide Möglichkeiten nicht gefallen. Mein Quellcode kann recht umfangreich werden, weshalb ich meine Methoden für den menschlichen Leser logisch gruppieren möchte (unabhängig vom Compiler, der den Quellcode linear durchgeht). Andererseits sind diese Methoden nicht Teil des öffentlichen Interface. Ich möchte sie nicht in der Header-Datei deklarieren. Was ich wirklich möchte, ist sie privat zu deklarieren.

Die Klassenerweiterung dient genau diesem Zweck. Nehmen Sie die folgende Zeile am Anfang Ihrer Implementierungsdatei direkt unter den Importen auf:

```
@interface BounceAppDelegate()
@end
```

Eine Klassenerweiterung sieht aus wie eine Kategorie ohne Namen. Weil die Erweiterung in unserer Implementierungsdatei definiert ist, teilen wir diese Methodendeklarationen nicht mehr mit allen anderen. Es sind private Funktionen, die nur wir selbst verwenden. Deklarieren Sie die neuen Methoden in der Klassenerweiterung, und sie können die Methoden überall innerhalb der Klassenimplementierung verwenden.

Categories/Bounce4/BounceAppDelegate.m

```
#import "BounceAppDelegate.h"
#import "NSSet+Persistence.h"

@interface BounceAppDelegate()
-(void)createSetOnDisk;
-(void)setFromDisk;
@end

@implementation BounceAppDelegate
@synthesize window;
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    [self createSetOnDisk];
    [self setFromDisk];
}
-(void) createSetOnDisk {
    NSSet *source = [NSSet setWithObjects:@"One",@"Two", nil];
     [source writeToFile:@"savedSet" atomically:YES];
}
-(void) setFromDisk {
    NSSet *fromDisk = [NSSet setWithContentsOfFile:@"savedSet"];
    NSLog(@"Set from disk: %@",fromDisk);
}
@end
```

Unsere Klassenerweiterung wird als Erweiterung der Methoden betrachtet, die im Klasseninterface deklariert sind. Und der Compiler zwingt Sie dazu, die darin deklarierten Methoden zu implementieren.[1]

23.5 Übung: Eigenschaften über Klassenerweiterungen erweitern

Hier ein andere schöne Sache, die man mit Klassenerweiterungen anstellen kann: Sie können eine Eigenschaft aufbauen, die von der Außenwelt nur gelesen werden kann, während Sie selbst sie lesen und schreiben können. Auf diese Weise können Sie alle Vorteile der Speicherverwaltung nutzen, die es bei Eigenschaften gratis dazu gibt, während der Clientcode nur den Wert der Eigenschaft abrufen darf.

Fügen Sie eine Nur-lese-Eigenschaft namens `retrievedSet` vom Typ `NSSet` in `BounceAppDelegate` ein. Fügen Sie am Ende der Implementierung von `application-DidFinishLaunching:` eine Zeile ein, die den Wert von `retrievedSet` in der Konsole ausgibt. Dazu verwenden Sie die Getter-Methode der Eigenschaft.

1 Bill Bumgarner hat in *http://www.friday.com/bbum/2009/09/11/class-extensions-explained/* über Implementierungsaspekte und Sinn und Zweck von Klassenerweiterungen gebloggt.

Ändern Sie `setFromDisk` so ab, dass sie `retrievedSet` mit dem Inhalt der Datei `savedSet` füllt. Sie verwenden den Setter, weshalb Sie bei der Kompilierung die folgende Fehlermeldung erhalten.

```
Object cannot be set -either readonly property or no setter found
```

Korrigieren Sie diesen Fehler, indem Sie eine Eigenschaftsdeklaration in die Klassenerweiterung einfügen. Sie erhalten vom Compiler die Warnung, dass diese Definition nicht mit der Eigenschaftsdefinition im Klasseninterface übereinstimmt. Diese Warnung ist korrekt. Es ist gut, dass man vor so etwas gewarnt wird, damit man nicht versehentlich das in der Header-Datei deklarierte Verhalten ändert. Der Build ist erfolgreich, und Sie können die Anwendung ausführen.

23.6 Lösung: Eigenschaften über Klassenerweiterungen erweitern

Fügen Sie die Instanzvariable und die Nur-lese-Eigenschaft in `BounceAppDelegate.h` ein:

Categories/Bounce5/BounceAppDelegate.h

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

@interface BounceAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
►   NSSet *retrievedSet;
}

► @property (readonly) NSSet *retrievedSet;
@property (assign) IBOutlet NSWindow *window;
@end
```

Synthetisieren Sie die Eigenschaft in der Implementierungsdatei und fügen Sie die folgende Zeile in die `applicationDidFinishLaunching:`-Methode ein:

Categories/Bounce5/BounceAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    [self createSetOnDisk];
    [self setFromDisk];
►   NSLog(@"Set from disk: %@", self.retrievedSet);
}
```

Die zusätzliche Zeile nutzt den Getter der Eigenschaft, und die folgende Änderung an `set-FromDisk` verwendet ihren Setter:

Categories/Bounce5/BounceAppDelegate.m

```
-(void) setFromDisk {
    self.retrievedSet = [NSSet setWithContentsOfFile:@"savedSet"];
}
```

Versehen Sie die Eigenschaft mit Schreibrechten, indem Sie die Deklaration von `retrievedSet` überschreiben:

Categories/Bounce5/BounceAppDelegate.m

```
@interface BounceAppDelegate()
-(void)createSetOnDisk;
-(void)setFromDisk;
► @property(copy) NSSet *retrievedSet;
@end
```

Nun betrachtet jeder Clientcode, der diesen Header importiert, die Eigenschaft als Nur-lese-Eigenschaft, während wir die Eigenschaft intern mit Lese- und Schreibrechten redefiniert haben, bevor die Methoden synthetisiert wurden. Sie können keine neue Eigenschaft in die Klassenerweiterung einfügen, wenn es nicht bereits eine zugrunde liegende Instanzvariable gibt. Hier haben wir nur die Attribute einer existierenden Eigenschaft geändert.

23.7 Kategorien und Core Data

Nehmen wir an, ich möchte meine Core Data-Entitäten um ein bestimmtes Verhalten erweitern. Nehmen wir an, ich möchte eine Art Bericht im Konsolenfenster ausgeben. Wir können eine Aktion in `CDBookshelf_AppDelegate.h` einfügen:

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

Fügen Sie einen Button in die Benutzerschnittstelle ein und verbinden Sie ihn mit dieser Aktion. Lassen Sie uns alle „managed Objects" im Speicher durchgehen und die auswählen, die wissen, wie man einen Report erstellt.

Categories/CDBookshelf8/CDBookshelf_AppDelegate.m

```
- (IBAction)createReport:(id) sender {
    for(NSManagedObject* element in
                    [[self managedObjectContext] registeredObjects]){
         if ([element respondsToSelector:@selector(PP_report)]) {
         [element PP_report];
        }
    }
}
```

Wo sollen wir die Methode `PP_report` generieren? In diesem ersten Beispiel wollen wir eine Kategorie zu `NSManagedObject` hinzufügen und sie dort anlegen. Deklarieren Sie sie im Header.

Categories/CDBookshelf8/NSManagedObject+Report.h

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

@interface NSManagedObject(Report)
-(void) PP_report;
@end
```

Die Implementierung gibt einfach die `description` an die Konsole aus:

Categories/CDBookshelf8/NSManagedObject+Report.m

```
#import "NSManagedObject+Report.h"

@implementation NSManagedObject(Report)
-(void)PP_report {
    NSLog(@"%@",[self description]);
}
@end
```

Sie können den Import für die Kategorie in ihren App-Delegate einfügen und *Build & Run* anklicken. Klicken Sie den *Report*-Button an, und Sie sehen eine Reihe von Einträgen wie den folgenden in der Konsole:

```
<NSManagedObject: 0x2000a9f60> (entity: Author;
id: 0x20008cb80 <x-coredata://44E-0D0D-46A2-9497-182BB3C/Author/p103> ;
data: {
    books = "<relationship fault: 0x2000c44e0 'books'>";
    firstName = Chad;
    lastName = Fowler;
    middleInitial = nil;
})
```

Das war ziemlich einfach, wenn man alle Objekte ausgeben will, egal von welcher Entität sie kommen. Doch was tun, wenn sich der Bericht auf Bücher beschränken soll? Wir lassen die Kategorie für `NSManaged-Object` verschwinden und führen stattdessen eine für `PragBook` ein.

23.8 Generierte Klassen in Core Data

Legen Sie eine neue Kategorie namens `Report` für die Klasse `PragBook` an. Die Implementierungsdatei sieht so aus:

Categories/CDBookshelf9/PragBook+Report.m

```
#import "PragBook+Report.h"

@implementation PragBook(Report)
-(void)PP_report {
    NSLog(@"%@",[self description]);
}
@end
```

Die Header-Datei muss PragBook importieren. Und das ist etwas, was wir nicht können.

Categories/CDBookshelf9/PragBook+Report.h

```
#import <Cocoa/Cocoa.h>
#import "PragBook.h"

@interface PragBook(Report)
-(void)PP_report;
@end
```

PragBook.h existiert nicht. Momentan heißt die mit der PragBook-Entität verknüpfte Klasse NSManagedObject. Wir müssen den Quellcode der PragBook-Klasse erzeugen.

Wählen Sie Ihr Datenmodell und dann im Menü File > New File... Wenn Sie das tun, während Ihr Datenmodell ausgewählt ist, erscheint eine neue Option unter Mac OS X > Cocoa Class. Wählen Sie Managed Object Class und klicken Sie den *Next*-Button an. Der Assistent erlaubt Ihnen die Auswahl von Lage, Projekt und Zielen. Bleiben Sie bei den Vorgaben und klicken Sie erneut den *Next*-Button an.

Auf der nächsten Seite können Sie wählen, für welche Entitäten Sie Klassen generieren. Wählen Sie die PragBook-Entität. Am unteren Rand der Seite stehen Ihnen drei Möglichkeiten zur Verfügung. Wählen Sie die „Generate Obj-C 2.0 Properties"-Box und lassen Sie „Generate accessors" und „Generate validation methods" unangetastet. Klicken Sie den *Finish*-Button an.

Das Datenmodell zeigt, dass die PragBook-Entität nun an die gerade generierte PragBook-Klasse gebunden ist und nicht mehr an das generische NSManagedObject:

Entity	Abs	Class
Author	☐	NSManagedObject
Chapter	☐	NSManagedObject
PragBook	☐	PragBook

Als Nächstes sehen Sie sich die generierte Header-Datei PragBook.h an:

Categories/CDBookshelf9/PragBook.h

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

@interface PragBook : NSManagedObject{}

@end
```

Darin ist nicht viel zu sehen, außer dass die `PragBook`-Klasse `NS-ManagedObject` erweitert, und nicht einfach `NSObject`. Die Implementierungsdatei ist ebenfalls leer. Mehr ist nicht notwendig, um unsere überarbeitete Anwendung zu kompilieren und auszuführen.

Klicken Sie auf *Build & Run*. Sie erhalten wie zuvor einen Bericht, aber diesmal sind nur `PragBook`-Objekte enthalten, und keine `Chapter`- oder `Author`-Objekte.

`PragBook` ist eine Subklasse von `NSManagedObject`. Sie sollten sorgfältig die Übersicht in der Dokumentation durchlesen, die Hinweise auf das Subclassing von `NSManagedObject` enthält. Unterm Strich sollten Sie die meisten Methoden auf keinen Fall überschreiben, und bei den anderen besteht keine Notwendigkeit dazu. Die Dokumentation lässt Sie außerdem wissen, dass es kaum einen Grund gibt, eigene Akzessormethoden zu schreiben.

In den meisten Fällen werden Sie Ihre Klassen um zusätzliche Funktionen erweitern und nicht vorhandene Verhaltensweisen einfügen wollen. Diese Modifikationen sollten in einer Kategorie vorgenommen werden und nicht in der generierten Datei. Kategorien erlauben Ihnen, die Funktionalität einer Klasse auf mehrere Quelldateien zu verteilen. Im Fall von Core Data ist das besonders praktisch, da Sie alle lokalen Modifikationen überschreiben würden, wenn Sie Ihr Datenmodell ändern und Klassendateien regenerieren.

23.9 Auf Eigenschaften zugreifen

Okay, jetzt kommt etwas sehr Cooles. Wir wollen den Bericht so ändern, dass er nur die Buchtitel ausgibt, und nicht die anderen Sachen, die in der `description` zurückgegeben werden.

Ändern Sie die Implementierung von `PP_report` wie folgt:

Categories/CDBookshelf10/PragBook+Report.m

```
#import "PragBook+Report.h"

@implementation PragBook(Report)
-(void)PP_report {
    NSLog(@"%@",[self title]);
}
@end
```

Wenn ich nun *Build & Run* anklicke und einen Bericht erzeuge, sehe ich Folgendes in der Konsole:

```
Core Animation
The Passionate Programmer
Core Data
iPhone SDK Development
```

Das ist wunderbar. Ich kann auf die Eigenschaft `title` zugreifen, indem ich ihren Getter aufrufe, weil die Attribute in Core Data KVC unterstützen. Ich ziehe es vor, über die Punktnotation auf die Eigenschaft zuzugreifen. Dazu muss ich die `PragBook`-Klasse mit anderen Optionen neu generieren.[2]

23.10 Klassendateien aus Entitäten neu generieren

Wählen Sie wieder Ihr Datenmodell aus, erzeugen Sie eine neue `Mac OS X > Cocoa Class` und wählen Sie dann erneut `Managed Object Class`. Wir wollen erneut Dateien für `PragBook`, erzeugen, aber diesmal bleiben die Checkboxen „Generate accessors" und „Generate Obj-C 2.0 Properties" aktiv.

Wenn Sie *Finish* anklicken, erhalten Sie die Warnung, dass die Template-Dateien bereits existieren. Löschen Sie die alten und erzeugen Sie neue. Das zeigt uns einen wichtigen Grund dafür, dass wir `PP_report` in einer Kategorie deklarieren und implementieren, und nicht in den `PragBook`-Quelldateien. Unsere Arbeit in den Quelldateien würde durch diesen Prozess verloren gehen. Kategorien erlauben uns, unseren Code an einer Stelle zu speichern, in der der Generator keinen Code ablegt. Das ist ein weiterer Grund, unseren Code auf verschiedene Orte zu verteilen.

Werfen wir einen Blick auf die neue Header-Datei für `PragBook`:

Categories/CDBookshelf11/PragBook.h

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

@interface PragBook : NSManagedObject {}

@property (nonatomic, retain) NSString * title;
@property (nonatomic, retain) NSSet* authors;
@property (nonatomic, retain) NSSet* chapters;
@end

@interface PragBook (CoreDataGeneratedAccessors)
```

2 Ich würde all das nicht tun, wenn ich nur die Punktnotation aktivieren wollte. Ich will Ihnen ein paar zusätzliche Punkte im Zusammenhang mit Core Data und Kategorien vermitteln.

```
- (void)addAuthorsObject:(NSManagedObject *)value;
- (void)removeAuthorsObject:(NSManagedObject *)value;
- (void)addAuthors:(NSSet *)value;
- (void)removeAuthors:(NSSet *)value;

- (void)addChaptersObject:(NSManagedObject *)value;
- (void)removeChaptersObject:(NSManagedObject *)value;
- (void)addChapters:(NSSet *)value;
- (void)removeChapters:(NSSet *)value;
@end
```

Wow, da wurde ganz schön viel Code eingefügt. Am Anfang gibt es drei Eigenschaften. Eine entspricht dem `title`-Attribut und die beiden anderen entsprechen den `authors`- und `chapters`-Relationen.

Der untere Teil besteht aus einem informellen Protokoll – einem Kategorie-Interface ohne entsprechende Implementierung. Wir wollen keine dieser Methoden implementieren und können daher das gesamte informelle Protokoll löschen.

Die Implementierungsdatei enthält ein Schlüsselwort, das wir bisher noch nicht gesehen haben:

Categories/CDBookshelf11/PragBook.m

```
#import "PragBook.h"

@implementation PragBook
@dynamic title;
@dynamic authors;
@dynamic chapters;

@end
```

Statt die Eigenschaften zu synthetisieren, verwendet die Vorlage das Schlüsselwort `@dynamic`. Das weist den Compiler an, keine Getter und Setter zu synthetisieren. In diesem Fall zeigt das an, dass die Methodenimplementierungen zur Laufzeit bereitgestellt werden.[3]

Die drei Eigenschaften stehen uns nun zur Verfügung, wir können also die `PP_report`-Methode so abändern, dass sie die Punktnotation für den Zugriff auf die `title`-Eigenschaft verwendet.

3 Sie können `@dynamic` auch benutzen, wenn Sie die Implementierung der Getter und Setter selbst zur Verfügung stellen.

Categories/CDBookshelf11/PragBook+Report.m

```
#import "PragBook+Report.h"

@implementation PragBook(Report)
-(void)PP_report {
►    NSLog(@"%@", self.title);
}
@end
```

Klicken Sie auf *Build & Run*, und wenn Sie den Button zur Generierung des Reports anklicken, erscheinen die Buchtitel in der Konsole.

In diesem Kapitel haben wir Kategorien verwendet, um Code zu von Ihnen entwickelten Klassen hinzuzufügen, zu von Apple entwickelten Klassen und zu Klassen, die aus Core Data-Entitäten erzeugt wurden. Wir haben auch mit einer Kategorievariante gearbeitet den Klassenerweiterungen , um eine Klasse um private Methoden zu erweitern. Sie haben viele Möglichkeiten und Freiheiten Sie sollten nicht die erstbeste wählen.

Kapitel 24

Blöcke

Wenn es Arbeit zu erledigen gilt, rufen wir üblicherweise eine Methode auf, die weiß, wie diese Arbeit zu erledigen ist, und übergeben ihr die benötigten Daten. In diesem Kapitel sehen wir uns drei Fälle an, in denen wir der Methode auch einen Teil der Arbeit übergeben, die es zu erledigen gilt. Diese Arbeit wird in Form eines speziellen Objective-C-Objekts übergeben, das als *Block* bezeichnet wird. Ein Block ist ein Stück ausführbaren Programmcodes, das zusammen mit den Daten übergeben werden kann, wobei die Daten aus dem Geltungsbereich kopiert werden, der den Block enthält.

Wir werden uns drei Einsatzgebiete für Blöcke ansehen. Zuerst werfen wir einen Blick auf Wrapper – das ist Code, der ein bestimmtes Setup vornimmt, den Code in einem Block aufruft und dann möglicherweise aufräumt, wenn der Block abgearbeitet ist. Danach sehen wir uns an, wie gut Blöcke mit Kollektionen von Objekten zusammenarbeiten. Sie ermöglichen Ihnen, bestimmten Code auf jedes Element der Kollektion anzuwenden, und zwar unabhängig davon, wie Sie auf die Kollektion zugreifen. Zum Schluss sehen wir uns Callbacks an – die Möglichkeit, einen bestimmten Code auszuführen, wenn ein bestimmtes Ereignis eintritt.[1]

Wenn Sie in anderen Sprachen mit Blöcken, Closures, Lambdas oder auch Funktionszeigern gearbeitet haben, können Sie viel von diesem Wissen auch hier anwenden. Die Syntax und die genauen Angaben mögen anders sein, aber die Grundideen und allgemeinen Strategien sind die gleichen.

1 Außerdem werden wir uns dem Thema Blöcke und Nebenläufigkeit in Kapitel 26, *Dispatch-Queues*, auf Seite 425 widmen.

Offizielle Unterstützung von Blöcken

Blöcke stehen momentan nur unter Mac OS X Snow Leopard (und höher) zur Verfügung. Sie können Blöcke beim iPhone und früheren Mac OS X-Versionen nicht verwenden.

24.1 Die Notwendigkeit von Blöcken in Wrappern

Nehmen wir an, Sie wollen zwei Integerwerte addieren, die als `NSNumber` gespeichert sind. Legen Sie eine neue CocoaAnwendung namens SimpleCalc an, sowie die folgende `add:to:`-Methode in `SimpleCalcAppDelegate.m`.[2]

Blocks/SimpleCalc1/SimpleCalcAppDelegate.m

```
-(NSNumber *) add: (NSNumber *)x to:(NSNumber *)y{
    NSInteger xAsInt = [x integerValue];
    NSInteger yASInt = [y integerValue];
    NSInteger result = xAsInt + yASInt;
    return [NSNumber numberWithInteger:result];
}
```

Die beiden `NSNumber` werden in `NSInteger` umgewandelt und addiert. Die Summe wird dann wieder in eine `NSNumber` umgewandelt und zurückgegeben.

Nur sehr wenig von diesem Code hat etwas mit der tatsächlichen Berechnung zu tun. Um das zu verdeutlichen, legen wir eine neue Methode namens `multiply:by:` an, die zwei `NSNumber` miteinander multipliziert. Ein Großteil dieses Codes ist mit dem in `add:to:` identisch. Wir wandeln die beiden `NSNumber` in `NSInteger` um und müssen ein Ergebnis zurückliefern. Alles, was sich ändert, sind der Methodenname und die Operation für die beiden `NSInteger`.

Blocks/SimpleCalc2/SimpleCalcAppDelegate.m

```
► -(NSNumber *) multiply: (NSNumber *)x by:(NSNumber *)y {
      NSInteger xAsInt = [x integerValue];
      NSInteger yASInt = [y integerValue];
►     NSInteger result = xAsInt * yASInt;
      return [NSNumber numberWithInteger:result];
  }
```

2 Ich habe die `window`-Eigenschaft aus dem App-Delegate entfernt. Achten Sie bei den Projekteinstellungen darauf, dass die Garbage Collection aktiv ist.

Was ich eigentlich möchte, ist eine Methode, die es mir erlaubt, zwei NSNumber zu übergeben, sowie die auszuführende Operation. Diese neue Methode verlangt zwei NSNumber als erste Parameter, die wir genau wie vorhin bei add:to: und multiply:by: umwandeln. Unsere neue Methode besitzt noch einen dritten Parameter, der die Operation angibt, die mit den beiden NSInteger durchzuführen ist. Diese Information wird in einem Block festgehalten.

24.2 Einen Block deklarieren

Hier die Signatur der Methode, durch die wir add:to: und multiply:by: ersetzen wollen:

Blocks/SimpleCalc4/SimpleCalcAppDelegate.m

```
-(NSNumber *) combine:(NSNumber *)x
                 with:(NSNumber *)y
           usingBlock:(NSInteger (^)(NSInteger,NSInteger)) block
```

Bevor wir uns die Blockdeklaration ansehen, werfen wir noch einen Blick auf die Deklaration des ersten Parameters der combine:with:usingBlock:-Methode:

```
(NSNumber *)x
```

Wir geben den Typ von x in Klammern vor der Variablen an. Dieser Typ ist ein Zeiger auf eine NSNumber.

Der dritte Paramter deklariert eine spezielle Art von Objekt, die als *Block* bezeichnet wird. Ein Block ist im Wesentlichen eine Funktion, die Daten abgreift und kopiert, sobald die Ausführung den Punkt der Deklaration des Blocks erreicht. Ein Block kann ein oder mehrere Eingaben verwenden und einen Wert zurückliefern. Das Format zur Spezifikation eines Blocks als Methodenparameter sieht grundsätzlich so aus:

```
(rückgabe_typ (^) (parameterTyp1, parameterTyp2)) block_name
```

Das ^ bezeichnet einen Block. Davor steht der Typ, der von diesem Block zurückgeben wird. Dahinter stehen die Parametertypen dieses Blocks.

In unserem Beispiel ist der Block so deklariert:

```
( NSInteger (^) (NSInteger, NSInteger)) block
```

Unser Block wird mit anderen Worten über den Namen `block` angesprochen. Dabei handelt es sich um eine Funktion, die zwei `NSInteger` erwartet und ein `NSInteger` zurückgibt. Auf diese Weise kann der Compiler prüfen, ob der übergebene Block die richtige Signatur aufweist.

Später in diesem Kapitel werden wir einen Block als Variable deklarieren. Diese Form sieht ein wenig anders aus. Unser Beispielblock würde dann so aussehen:

```
NSInteger (^block) (NSInteger, NSInteger) = // Block-Definition
```

Wir nutzen die gleiche Form, wenn unser Block als Parameter für eine C-Funktion dient:

```
(NSInteger *) combineUsingBlock (NSNumber *x, NSNumber *y,
                                 NSInteger (^block)(NSInteger,NSInteger)){
//Funktionsrumpf
}
```

Wir werden hier aber die Methodenparameter-Version weiternutzen. Als Nächstes werden Sie sehen, wie man einen bestimmten Block implementiert und aufruft.

24.3 Blöcke in Wrappern nutzen

Sie nutzen den Block innerhalb der `combine:with:usingBlock:`-Methode, als wäre er eine Funktion.

Blocks/SimpleCalc4/SimpleCalcAppDelegate.m

```
-(NSNumber *) combine:(NSNumber *)x
                 with:(NSNumber *)y
           usingBlock:(NSInteger (^)(NSInteger,NSInteger)) block
{
    NSInteger xAsInt = [x integerValue];
    NSInteger yASInt = [y integerValue];
►   NSInteger result = block(xAsInt, yASInt);
    return [NSNumber numberWithInteger:result];
}
```

Wir sind so weit, einen Block in der `combine:with:usingBlock:`-Methode nutzen zu können. Hier sehen Sie den Block, den wir an diese Methode übergeben, um zwei Zahlen zu addieren. In diesem Fall kann der Typ des Blocks abgeleitet werden. Sie müssen ihn daher nicht angeben.

```
^(NSInteger x, NSInteger y) {
    return x+y;
}
```

Diese Art Ausdruck wird *Blockliteral* genannt. Der Rumpf kann mehrere Anweisungen und deklarierte Variablen enthalten. Hier ist das Ganze sehr einfach: Es handelt sich um einen Block, der zwei `NSInteger` als Argumente verlangt. Wir müssen diese Argumente benennen, damit wir sie im Rumpf des Blocks nutzen können. Wir nennen Sie `x` und `y`, und der Block macht nichts weiter, als ihre Summe zurückzugeben.

Wenn Sie also `block(xAsInt,yAsInt)` in `combine:with:usingBlock:` aufrufen, entspricht das dem Aufruf dieser Funktion:

```
(NSInteger) block(NSInteger x, NSInteger y) {
    return x+y;
}
```

Die Stärke von Blöcken liegt darin, dass Sie den Rumpf der Funktion durch die Übergabe eines anderen Blocks so ändern können, dass sie das Produkt zurückliefert. Hier die komplette `applicationDidFinish-Launching:`-Methode. Ich habe die Übergabe der verschiedenen Blöcke entsprechend hervorgehoben.

Blocks/SimpleCalc4/SimpleCalcAppDelegate.m

```
1  -(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
2    NSNumber *firstNumber = [NSNumber numberWithInteger:7];
3    NSNumber *secondNumber = [NSNumber numberWithInteger:5];
4    NSNumber *sum = [self combine:firstNumber
5                             with:secondNumber
►                     usingBlock:^(NSInteger x,NSInteger y){return x+y;}];
7    NSLog(@"The sum of %@ and %@ is %@.",
8          firstNumber, secondNumber, sum);
9    NSNumber *product = [self combine:firstNumber
10                            with:secondNumber
►                      usingBlock:^(NSInteger x,NSInteger y){return x*y;}];
12   NSLog(@"The product of %@ and %@ is %@.",
13         firstNumber, secondNumber, product);
14 }
```

Sie wissen bereits, wie man einen Block als Parameter übergibt. Wir übergeben die Blöcke für die Addition und Multiplikation in den Zeilen 6 und 11.

Das war ein ziemlich albernes Beispiel für den Einsatz von Wrappern, aber es illustriert die Technik. Ein etwas typischerer Anwendungsfall wäre die Durchführung von Aktionen bei einer entfernt gespeicherten Datei. Unabhängig von der eigentlichen Aktion müssen Sie vor dieser Aktion immer eine Reihe von Schritten abarbeiten, um auf die Datei zugreifen zu können, und nach der Aktion sind bestimmte Schritte notwendig, um die Ressourcen aufzuräumen. Es erscheint durchaus sinnvoll, diese Methode, die mittendrin dieses Protokoll mit der akzeptierten

Aktion enthält, als Block auszulegen. Tatsächlich sollten Sie jedes Mal, wenn einer Aktion die gleichen Schritte vorangehen und/oder folgen, die Wrapper-Technik mit Blöcken in Erwägung ziehen.

Wie sieht es mit der Iteration über ein Array und der Durchführung einer Aktion für jedes seiner Elemente aus? Das ist ein so gängiger Fall, dass Apple die Möglichkeit eingebaut hat, einen Block an jedes Element einer Kollektion in Arrays, Sets und Dictionaries zu senden. Wir sehen uns das an, nachdem wir gezeigt haben, wie Blöcke mit Variablen arbeiten, die im selben Geltungsbereich definiert sind.

24.4 Werte abfangen

Bisher haben wir ein wesentliches Feature von Blöcken ignoriert: Sie enthalten einen Schnappschuss der Werte von Variablen, die bei der Deklaration des Blocks im Geltungsbereich vorhanden sind. Ich werde unser Beispiel auf verschiedene Weise vereinfachen. Zuerst werde ich die `NSNumber` durch `NSInteger` ersetzen, um die ständige Konvertierung zu eliminieren. Wir haben damit gezeigt, wie und warum man Blöcke nutzt, um eine Operation in einen Wrapper zu übergeben, aber jetzt stehen sie uns eher im Weg. Darüber hinaus wollen wir nicht einfach zwei Zahlen miteinander multiplizieren, sondern einen Wert verwenden, der in einer lokalen Variable definiert ist, die in dem Geltungsbereich liegt, in dem auch der Block definiert ist.

Blocks/SimpleCalc5/SimpleCalcAppDelegate.m

```
1  #import "SimpleCalcAppDelegate.h"
2
3  @implementation SimpleCalcAppDelegate
4
5  -(NSInteger) tripleUsingBlock:(NSInteger (^)(NSInteger)) block
6  {
7      return block(3);
8  }
9  -(void)applicationDidFinishLaunching:(NSNotification *)aNotification {
10     NSInteger multiplicand = 5;
11     NSInteger product = [self tripleUsingBlock:^(NSInteger multiplier){
12         return multiplier * multiplicand;
13     }];
14     NSLog(@"Triple %d is %d.", multiplicand, product);
15 }
16 @end
```

In diesem kurzen Listing passiert eine ganze Menge. Wir beginnen mit der Deklaration der `tripleUsingBlock:`-Methode in Zeile 5. Wir übergeben keinen Wert für den Multiplikanden (`multiplicand`). Der einzige Parameter ist der Block selbst.

Der `multiplicand` ist in `applicationDidFinishLaunching:` deklariert, direkt vor der Deklaration des Blocks in Zeile 11. Hier die eigentliche Definition des Blocks:

```
^(NSInteger multiplier){ return multiplier * multiplicand; }
```

Der Block erwartet als einziges Argument ein `NSInteger` namens `multiplier`. Jetzt kommt der springende Punkt: Der Block gibt das Ergebnis der Multipikation dieses `multiplier` mit der zweiten Variablen namens `multiplicand` zurück. Der `multiplicand` weist zum Zeitpunkt der Ausführung von `^` den Wert `5` auf, dieser Wert wandert also mit in den Block. Wenn der Block in Zeile 10 aufgerufen wird, wird der Wert des `multiplicand` bei `5` eingefroren, und dieser Wert wird mit `3` multipliziert.

Wo Sie jetzt verstehen, was genau vor sich geht, sehen Sie sich den Code noch einmal an. Zunächst sehen Blöcke ein wenig fremdartig aus. Wenn man sich das Codelisting jetzt so ansieht, fangen sie an, etwas natürlicher auszusehen. Als Nächstes wollen wir dieses Beispiel erweitern und Blöcke an Kollektionen übergeben.

24.5 Blöcke und Kollektionen

Apple stellt in Snow Leopard Dutzende von Methoden zur Verfügung, mit deren Hilfe Sie Blöcke an Kollektionen übergeben können, um ihre Elemente zu sortieren oder zu transformieren. Als Beispiel für ihre Anwendung wollen wir einige Integerwerte in einem `NSArray` ablegen und dieses Array dann durchgehen und jedes Element mit `3` multiplizieren.

Legen Sie ein neues Cocoa-Projekt namens *CollectionCalc* an. In `applicationDidFinishLaunching:` erzeugen wir unser Array und geben es aus. Dann multiplizieren wir jedes Element mit `3` und geben das Ergebnis aus.

Blocks/CollectionCalc1/CollectionCalcAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *numbers = [self createArray];
    NSLog(@"Elements in the initial array:%@", numbers);
    NSArray *transformedNumbers = [self tripleElementsIn: numbers];
    NSLog(@"Elements in the tripled array:%@", transformedNumbers);
}
```

Zur `createArray`-Methode gibt es nicht viel zu sagen. Sie müssen nur daran denken, dass Arrays nur Objekte und keine Primitive enthalten dürfen, dass wir also unsere `NSInteger` in `NSNumber` umwandeln müssen.

Blocks/CollectionCalc1/CollectionCalcAppDelegate.m

```
-(NSArray *) createArray {
    return [NSArray arrayWithObjects:[NSNumber numberWithInt:5],
         [NSNumber numberWithInt:2],
         [NSNumber numberWithInt:17],
         [NSNumber numberWithInt:-3],
         [NSNumber numberWithInt:14],nil];
}
```

Wir transformieren das Array in der Methode `tripleElementsIn:`. Wenn wir nicht mit Blöcken arbeiten würden, käme die schnelle Enumerierung mit `for in` zum Einsatz, die in Objective-C 2.0 eingeführt wurde. In diesem Fall wären wir für das Durchlaufen des Arrays verantwortlich. Nachdem sich der Enumerator zum nächsten Element bewegt hat, fragt er uns, welche Arbeit als Nächstes erledigt werden soll.

```
for (NSNumber *element in originalArray) {
    // Verarbeitung erfolgt hier
}
```

Bei Snow Leopard können wir diese Methode für ein Array aufrufen:

```
- (void)enumerateObjectsUsingBlock:
             (void (^)(id obj, NSUInteger idx, BOOL *stop))block
```

Nun weisen wir das Array an, den von uns übergebenen Block auf jedes seiner Elemente anzuwenden. Die Enumerierung ist hier keine separate Aktion. Wir wollen diese Technik nutzen, um jeden Wert des Arrays mit 3 zu multiplizieren:

Blocks/CollectionCalc1/CollectionCalcAppDelegate.m

```
-(NSArray *) tripleElementsIn:(NSArray *) originalArray {
   NSMutableArray *tempArray =
   [[NSMutableArray alloc] initWithCapacity:[originalArray count]];
►  [originalArray enumerateObjectsUsingBlock:
►                          ^(id obj, NSUInteger idx, BOOL *stop)       {
►    [tempArray addObject:[NSNumber numberWithInt: 3 * [obj intValue]]];
►  }];
   return tempArray;
}
```

Beachten Sie, dass in diesem Array-Beispiel drei Parameter an den Block übergeben werden. Der erste Parameter `obj` ist der Zeiger auf das aktuelle Element. Das gibt uns ein Handle auf das Objekt, dessen Wert wir verdreifachen wollen. Der zweite Parameter ist der Index des aktuellen Elements. In unserem Beispiel haben wir dafür keine Verwendung.

Der letzte Parameter fungiert als `break`. Dieser `BOOL`-Wert bricht die Iteration über das Array ab, sobald er auf `YES` gesetzt wird. Sie könnten beispielsweise ein Array durchlaufen wollen, bis Sie das erste Vorkommen von etwas finden. An diesem Punkt setzen Sie `stop` auf `YES`.

An dieser Stelle fragen Sie sich wahrscheinlich, was daran so toll sein soll. Gute Frage. Graben wir ein wenig tiefer. Erinnern Sie sich daran, dass Blöcke Objekte sind. Wir können Sie deklarieren, initialisieren und da- und dorthin übergeben, statt sie an einem festen Ort zu deklarieren. Sehen wir uns an, wie das Ihren Code verändert.

24.6 Blöcke deklarieren, definieren und benutzen

Bisher haben wir unsere Blöcke *inline* angegeben. Wenn Blöcke sehr kurz sind, ist das eine bequeme Möglichkeit, um die notwendige Arbeit zu erledigen. Doch Blöcke sind auch Objekte, wir können sie also deklarieren und benutzen wie alle anderen Objekte auch. Lassen Sie uns zum Beispiel einen Block deklarieren, der eine `NSNumber` und eine `NSInteger` addiert und das Produkt als `NSNumber` zurückgibt. Wir legen die Instanzvariable und die zugehörige Eigenschaft in `CollectionCalcAppDelegate.h` an.

Blocks/CollectionCalc2/CollectionCalcAppDelegate.h

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

@interface CollectionCalcAppDelegate : NSObject <NSApplicationDelegate> {
        NSNumber *(^multiply)(NSNumber *, NSInteger);
}
@property(copy) NSNumber *(^multiply)(NSNumber *, NSInteger);
@end
```

Vergessen Sie nicht, diese Eigenschaft in `CollectionCalcAppDelegate.m`: zu synthetisieren:

Blocks/CollectionCalc2/CollectionCalcAppDelegate.m

```
@synthesize multiply;
```

Fügen Sie die Blockinitialisierung in die `applicationDidFinishLaunching:`-Methode ein:

Blocks/CollectionCalc2/CollectionCalcAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *numbers = [self createArray];
►       self.multiply = ^(NSNumber *x, NSInteger y) {
►               return [NSNumber numberWithInt:[x intValue] * y];
►       };
    NSLog(@"Elements in the initial array:%@", numbers);
    NSArray *transformedNumbers = [self tripleElementsIn: numbers];
    NSLog(@"Elements in the tripled array:%@", transformedNumbers);
}
```

Wird ein Block außerhalb seines Geltungsbereichs übergeben, müssen Sie ihn kopieren. Andernfalls könnte der Block freigegeben werden, wenn die Methode zurückkehrt, die die Blockinitialisierung enthält. Aus diesem Grund benutzen wir die Eigenschaft für `multiply`. Wir nutzen den Vorteil des Speicherattributs `copy`, den wir für `multiply` gesetzt haben.

Nun nutzen wir diesen Block innerhalb des Blocks, den wir zur Enumerierung des Arrays verwendet haben:

Blocks/CollectionCalc2/CollectionCalcAppDelegate.m

```
-(NSArray *) tripleElementsIn:(NSArray *) originalArray {
     NSMutableArray *tempArray =
     [[NSMutableArray alloc] initWithCapacity:[originalArray count]];
     [originalArray enumerateObjectsUsingBlock:
                                ^(id obj, NSUInteger idx, BOOL *stop) {
►          [tempArray addObject:multiply(obj,3)];
    }];
    return tempArray;
}
```

Ist das nicht schön? Wenn ich die Zeilen für dieses Buch nicht hätte umbrechen müssen, hätten wir das gesamte Array in einer einzige Zeile transformiert.

24.7 Die Verwendung von __block

Wir werden dieses Beispiel in verschiedene Richtungen ausweiten, um Variablen zu untersuchen. In diesem Abschnitt wollen wir eine neue Methode namens `squareElementsIn:` entwickeln, die das Quadrat jedes Elements des Original-Arrays zurückgibt.

Blocks/CollectionCalc3/CollectionCalcAppDelegate.m

```
-(NSArray *) squareElementsIn:(NSArray *) originalArray {
    NSMutableArray *tempArray =
    [[NSMutableArray alloc] initWithCapacity:[originalArray count]];
    [originalArray enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop) {
►                [tempArray addObject:multiply(obj,[obj intValue])];
        }];
    return tempArray;
}
```

Rufen Sie die Methode in `applicationDidFinishLaunching:` auf und geben Sie das Ergebnis aus. Wenn Sie *Build & Run* anklicken, läuft alles prima und Sie erhalten ein Array mit dem Quadrat des ursprünglichen Arrays.

Nun führen wir eine temporäre Variable für `[obj integerValue]` außerhalb des Blocks ein, damit Sie sehen, was schiefgehen kann.

Blocks/CollectionCalc4/CollectionCalcAppDelegate.m

```
-(NSArray *) squareElementsIn:(NSArray *) originalArray {
    NSMutableArray *tempArray =
    [[NSMutableArray alloc] initWithCapacity:[originalArray count]];
►       NSInteger multiplier;
    [originalArray enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop) {
►                multiplier = [obj intValue];
►                [tempArray addObject:multiply(obj,multiplier)];
        }];
    return tempArray;
}
```

Das wird nicht einmal kompiliert. Wir haben eine `NSInteger` außerhalb des Blocks deklariert und versucht, sie zu aktualisieren. Die Fehlermeldung lautet „Assignment of read-only variable 'multiplier'." Wir hätten dieses Problem vermeiden können, indem wir die Variable innerhalb des Blocks deklariert hätten, aber wir wollen verstehen lernen, warum `multiplier` „read-only" ist.

Das ist ein sehr wichtiger Punkt. Wenn Sie in den Block eintreten, legt die Laufzeitumgebung einen Schnappschuss der zu dieser Zeit zugänglichen Variablen an. Aus diesem Grund konnten wir die `multiplicand`-Variable in dem Block nutzen, den wir in Abschnitt 24.4, *Werte abfangen*, auf Seite 400 verwendet haben. Solange nichts anderes angegeben ist, werden diese Variablen behandelt wie Konstanten – der Code im Block kann ihre Werte lesen, aber nicht verändern. Das ist gut für die Performance und für Multithreading.

Genauer gesagt, wird die Ausführung beim Eintritt in den Block an das Blockliteral übergeben. Die Werte von Nicht-Objekt-Typen werden zu diesem Zeitpunkt gesetzt. Wäre `multiplier` kein `NSInteger`, sondern ein Objekt, könnte das Objekt, auf das die Variable verweist, durch den Block verändert werden.

Wie kann man das nun lösen? Wenn ein Block den Wert einer Nicht-Objekt-Variablen ändern muss, die außerhalb des Blocks deklariert ist, müssen Sie `__block` verwenden, wenn Sie die Variable deklarieren.[3] Mit dieser kleinen Änderung wird die Anwendung fehlerfrei kompiliert und ausgeführt.

Blocks/CollectionCalc5/CollectionCalcAppDelegate.m

```
-(NSArray *) squareElementsIn:(NSArray *) originalArray {
     NSMutableArray *tempArray =
     [[NSMutableArray alloc] initWithCapacity:[originalArray count]];
►        __block NSInteger multiplier;
     [originalArray enumerateObjectsUsingBlock:
        ^(id obj, NSUInteger idx, BOOL *stop) {
                multiplier = [obj intValue];
                [tempArray addObject:multiply(obj,multiplier)];
        }];
    return tempArray;
}
```

Bei Blöcken und Variablen gibt es viele Feinheiten zu beachten. Sie sollten definitiv Apples *Blocks Programming Topics* [App09b] lesen, insbesondere den Abschnitt „Blocks and Variables."

24.8 Aufräumen mit typedef

Lassen Sie uns einen zweiten Block namens `add` deklarieren, der die gleiche Signatur hat wie `multiply`. Deklarieren Sie ihn als Eigenschaft und synthetisieren sie ihn. Wir werden diesen Block nicht benutzen. Ich will Ihnen nur eine Möglichkeit zeigen, Ihre Deklarationen aufzuräumen.

Blocks/CollectionCalc6/CollectionCalcAppDelegate.h

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

@interface CollectionCalcAppDelegate : NSObject <NSApplicationDelegate> {
        NSNumber *(^multiply)(NSNumber *, NSInteger);
        NSNumber *(^add)(NSNumber *, NSInteger);
}
@property(copy) NSNumber *(^multiply)(NSNumber *, NSInteger);
@property(copy) NSNumber *(^add)(NSNumber *, NSInteger);
@end
```

3 Das sind zwei Unterstriche und das Wort `block`.

Die add- und multiply-Blöcke sind vom selben Typ – sie erwarten beide eine NSNumber und ein NSInteger und geben eine NSNumber zurück. Wir wollen dieses Konzept formalisieren, indem wir typedef nutzen, um diesen neuen Typ zu definieren, den wir ArithmeticOperation nennen werden.

Blocks/CollectionCalc7/CollectionCalcAppDelegate.h

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

typedef NSNumber *(^ArithmeticOperation)(NSNumber *, NSInteger);

@interface CollectionCalcAppDelegate : NSObject <NSApplicationDelegate> {
       ArithmeticOperation multiply;
       ArithmeticOperation add;
}
@property(copy) ArithmeticOperation multiply;
@property(copy) ArithmeticOperation add;
@end
```

Dieser Ansatz hat Vor- und Nachteile. Auf der Habenseite können wir verbuchen, dass der Code wesentlich klarer ist. Negativ könnte sich bei großen Programmen auswirken, dass andere mühsam die verschiedenen typedef aufspüren und herausfinden müssen, was da genau vor sich geht.

24.9 Übung: Blöcke in Callbacks benutzen

Wir wollen diese Einführung in Blöcke mit einem Beispiel für ein Callback abschließen. Unsere Anwendung wird alle Notifikationen des Notification Centers abfangen und in der Konsole ausgeben. Wir müssen eine Nachricht an das Notification Center senden und das Objekt sowie die Methode angeben, die vom Notifcation Center aufgerufen werden soll, wenn diese Art Notifikation eintritt. Sobald wir das auf traditionelle Weise eingerichtet haben, werden wir den Callback-Mechanismus durch einen Block ersetzen.

Legen Sie eine neue Cocoa-Anwendung an, die Core Data nicht nutzt. Nennen Sie sie Callback und ändern Sie CallbackAppDelegate.m wie folgt:

Blocks/Callback1/CallbackAppDelegate.m

```
#import "CallbackAppDelegate.h"

@implementation CallbackAppDelegate

@synthesize window;
```

```
-(void) response:(NSNotification *) notification {
    NSLog(@"Received: %@.", [notification name]);
}
-(void)registerWithoutBlocks {
     [[NSNotificationCenter defaultCenter]
                                        addObserver:self
                                           selector:@selector(response:)
                                               name:nil
                                             object:nil];
}
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
             [self registerWithoutBlocks];
}
@end
```

Sehen Sie sich die `registerWithoutBlocks`-Methode an, wo wir uns für den Empfang von Notifikationen registrieren. In diesem Fall übergeben wir `nil` als Notifikationsnamen, um alle Notifikationen zu empfangen. Wir geben auch an, welches Objekt benachricht werden soll und welche Methode als Callback dient, wenn die Notifikation empfangen wird.

Sie können erkennen, dass die Callback-Methode `response:` nicht besonders viel tut.[4]

Die Snow Leopard-APIs kennen eine neue Methode zur Registrierung eines Blocks zum Empfang von Notifikationen: Man eliminiert die Notwendigkeit einer Callback-Methode, indem man das, was beim Eingang der Notifikation geschehen soll, in einem Block übergibt. Diese eine Methode kombiniert die Notifikationsregistrierung mit dem Callback.

```
addObserverForName:(NSString *)name
            object:(id)obj
                 queue:(NSOperationQueue *)queue
            usingBlock:(void (^)(NSNotification *arg1))block
```

Ändern Sie `CallbackAppDelegate.m` so ab, dass es diese Methode und einen Block nutzt.

24.10 Lösung: Blöcke in Callbacks nutzen

Sie können die `response:`-Methode eliminieren und die `registerWithoutBlocks`-Methode durch folgende ersetzen:

4 Ich empfinde es als störend, dass man wissen muss, dass die Callback-Methode einen einzigen Parameter vom Typ `NSNotication` verlangt. Wie Sie sehen werden, ist diese Forderung in der Blockversion deutlicher zu erkennen.

Blocks/Callback2/CallbackAppDelegate.m

```
-(void)registerWithBlocks {
     [[NSNotificationCenter defaultCenter]
                      addObserverForName:nil
                                  object:nil
                                   queue:nil
►                           usingBlock:^(NSNotification *notification) {
►                           NSLog(@"Received: %@.", [notification name]);
►                           }];
}
```

Der Rumpf des Blocks ist der Rumpf der alten `response:`-Methode. Der Klarheit halber habe ich den Namen der `NSNotication` von `arg1` in `notication` geändert. Ein Vorteil der Verwendung von Blöcken liegt darin, dass Sie das, was getan werden muss, dort ausdrücken, wo es getan werden muss: Ihre Logik verteilt sich nicht mehr überall.

Klicken Sie auf *Build & Run*, und die Konsole sollte sich mit Notifikationen füllen:

```
Received: NSMenuDidChangeItemNotification.
Received: NSWindowDidBecomeKeyNotification.
Received: NSWindowDidBecomeMainNotification.
```

Es gibt ein großes Problem mit diesem Code. Das Problem hat nichts mit Blöcken zu tun, aber so viele Leute stolpern darüber, dass wir darüber reden sollten. Nach wenigen Sekunden hören die Notifikationen auf. Selbst wenn Sie die Größe des Fensters ändern, sehen Sie keine `NSWindowDidResizeNotication` oder andere verwandte Notifikationen.

Sie könnten sich an dieses Problem von unserer Arbeit mit KVO erinnern. Mit der automatischen Garbage Collection müssen wir explizit am Observer festhalten, oder er wird vom Garbage Collector freigegeben und wir haben kein Objekt mehr, das die Notifikationen empfangen könnte. Fügen Sie eine Instanzvariable in die Header-Datei ein.

Blocks/Callback3/CallbackAppDelegate.h

```
#import <Cocoa/Cocoa.h>
@interface CallbackAppDelegate :NSObject <NSApplicationDelegate> {
    NSWindow *window;
►   NSObject *observer;
}
@property (assign) IBOutlet NSWindow *window;
@end
```

Lassen Sie `observer` das Objekt sein, das zurückgegeben wird, wenn wir uns für den Empfang von Notifikationen registrieren, und erhalten Sie dieses Objekt mit *retain*.

Blocks/Callback3/CallbackAppDelegate.m

```
-(void)registerWithBlocks {
►   observer = [[[NSNotificationCenter defaultCenter]
                 addObserverForName:nil object:nil queue:nil
                         usingBlock:^(NSNotification *notification) {
                           NSLog(@"Received: %@.", [notification name]);
►                        }] retain];
}
```

Nun sollte alles funktionieren. Notifikationen werden gesendet und empfangen, bis die Anwendung beendet wird.

Blöcke stellen die Art und Weise auf den Kopf, wie Sie über Code denken. Statt eine Vielzahl von Parametern an eine Methode zu übergeben, die weiß, was sie damit anstellen muss, übergeben Sie dem Objekt eine Aktion zusammen mit einem Schnappschuss der Daten, mit denen gearbeitet werden muss. Mit etwas Erfahrung werden Sie ein Gefühl für die Fälle entwickeln, in denen Sie von Blöcken profitieren können.

Sie finden online viel gutes Material über Blöcke. BBum bietet eine Grundreihe von Postings an, die Sie in seinem weblog-o-mat lesen sollten. Lesen Sie insbesondere Basic Blocks[5] und Blocks Tips & Tricks.[6] Mike Ash hat zwei sehr gute Einträge in seiner freitäglichen Q&A-Serie zu Blöcken gepostet.[7] [8] Joachim Bengtsson bietet eine Online-Einführung in Blöcke an,[9] und Drew McCormack hat eine exzellente Anleitung in „10 Uses for Blocks" in seiner „Cocoa for Scientists"-Serie geschrieben. [10]

In diesem Kapitel haben Sie ein Gefühl dafür bekommen, wie Sie Blöcke auf verschiedene Weise nutzen können, um bestimmte Verhaltensweisen zu verpacken, ein Callback zu nutzen oder mit Kollektionen zu arbeiten. Als Nächstes wollen wir uns mit Nebenläufigkeit beschäftigen und damit, wie Sie Blöcke aufbauen, damit Ihr Code schneller ausgeführt wird.

5 http://www.friday.com/bbum/2009/08/29/basic-blocks/
6 http://www.friday.com/bbum/2009/08/29/blocks-tips-tricks/
7 http://mikeash.com/pyblog/friday-qa-2008-12-26.html
8 http://mikeash.com/pyblog/friday-qa-2009-08-14-practical-blocks.html
9 http://thirdcog.eu/pwcblocks/
10 http://www.macresearch.org/cocoa-scientists-xxxii-10-uses-blocks-cobjective-c

Kapitel 25

Operationen und ihre Queues

Nebenläufigkeit (concurrency) ist schwierig. Computer sind nicht besonders gut darin, den Zugriff auf gemeinsame Ressourcen (insbesondere Speicher) zu regeln. Die gute Nachricht ist, dass hart daran gearbeitet wird, die Lage zu vereinfachen, damit wir die zusätzlichen Prozessoren in unseren Maschinen zu unserem Vorteil nutzen können.

Eine der Techniken besteht darin, größere Aufgaben in diskrete Arbeitseinheiten aufzuteilen, die hintereinander oder parallel in separaten Threads ausgeführt werden können. Ist die Arbeit erledigt, sammelt das Hauptprogramm die Ergebnisse ein.

In Mac OS X Leopard wurde die Idee der Operation-Queues eingeführt, um die Verteilung der Arbeit auf mehrere Threads zu koordinieren. Snow Leopard geht mit Dispatch-Queues (die wir im nächsten Kapitel behandeln) noch einen Schritt weiter.

Wir beginnen mit einer mangelhaften Anwendung, die den Benutzer warten lässt (der berühmte rotierende Ball). Die Mängel beheben wir dann mithilfe von Operation-Queues.

25.1 Den Ball rotieren lassen

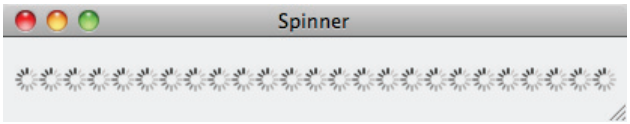
Im ersten Beispiel wollen wir 25 Fortschrittsanzeigen nacheinander für jeweils eine Sekunde rotieren lassen. Die Anwendung soll nicht weiter reagieren und Sie sollen den rotierenden Ball sehen, während die Fortschrittsanzeige dargestellt wird.

Legen Sie eine neue Cocoa-Anwendung namens *Spinner* an. Beim Start der Anwendung wird zuerst ein Array mit 25 Fortschrittsanzeigen erzeugt und konfiguriert. Danach wird jede für eine Sekunde aktiviert.

Operations/Spinner1/SpinnerAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *arrayOfSpinners = [self arrayOfSpinners];
    for (NSProgressIndicator *spinner in arrayOfSpinners) {
         [self spin:spinner];
    }
}
```

Ich lege die Größe des Fensters und die Position der Fortschrittsanzeigen so fest, dass es wie folgt aussehen würde, wenn alle Anzeigen gleichzeitig sichtbar wären:



Dazu mache ich das Fenster im Interface Builder 415 Pixel breit und 56 Pixel hoch. Erzeugen und platzieren Sie die Fortschrittsanzeigen mit Xcode in der `arrayOfSpinners`-Methode.

Operations/Spinner1/SpinnerAppDelegate.m

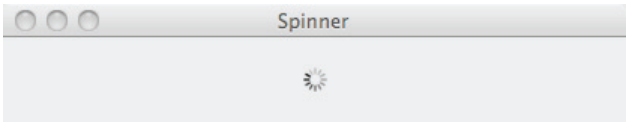
```
-(NSArray *) arrayOfSpinners {
    NSMutableArray *array = [[NSMutableArray alloc] initWithCapacity:25];
    for (int i =0;i < 25; i++){
        NSProgressIndicator *spinner = [[NSProgressIndicator alloc]
                    initWithFrame: NSMakeRect(16 * i + 8, 20, 16, 16)];
         [spinner setStyle:NSProgressIndicatorSpinningStyle];
         [spinner setControlSize:NSSmallControlSize];
         [spinner setDisplayedWhenStopped:NO];
         [window.contentView addSubview:spinner];
         [array addObject:spinner];
    }
    return array;
}
```

Die Methode `arrayOfSpinners` gibt ein Array zurück, das mit Fortschrittsanzeigen gefüllt ist. Wir gehen dieses Array durch, starten die Animation, warten eine Sekunde und halten die Animation an.

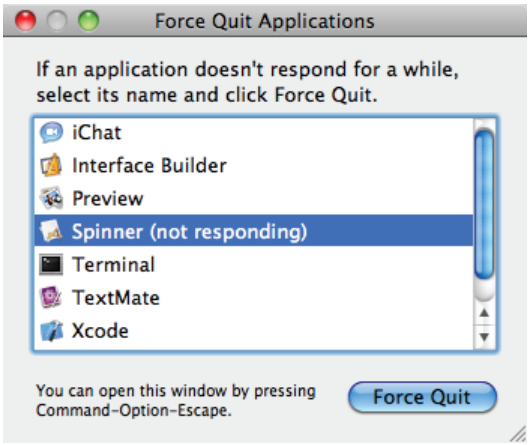
Operations/Spinner1/SpinnerAppDelegate.m

```
-(void) spin:(NSProgressIndicator *) spinner {
    [spinner startAnimation:self];
    sleep(1);
    [spinner stopAnimation:self];
}
```

Klicken Sie auf *Build & Run*. Die Anwendung startet, und die Fortschrittsanzeigen erscheinen und drehen sich nacheinander. Wenn Sie den Mauszeiger über das Fenster bewegen, sehen Sie den rotierenden Ball. Sie werden bemerken, dass das Fenster nicht das aktive Fenster ist. Wenn Sie versuchen, es anzuklicken, reagiert es nicht.



Sie können das auch beobachten, wenn Sie eine Release-Version der Anwendung erzeugen und sie als eigenständige Anwendung ausführen. Sie wird im „Sofort beenden..."-Fenster als nicht reagierend gemeldet.



Die Anwendung reagiert nicht, weil der Ereignisverarbeitungs-Thread blockiert, während er darauf wartet, dass die `spin`-Methode abgeschlossen wird. Wir wollen diese fehlende Ansprechbarkeit mit Operationen und Queues beheben. Es gibt drei grundlegende Arten von Operationen. Wir wollen mit einem Typ beginnen, der es uns erlaubt, aus einer Methode eine Operation zu machen.

25.2 Operationen aufrufen

Im Moment rufen wir die `spin:`-Methode direkt auf:

Operations/Spinner1/SpinnerAppDelegate.m

```
for (NSProgressIndicator *spinner in arrayOfSpinners) {
     [self spin:spinner];
}
```

Eine `NSInvocationOperation` erlaubt es uns, ein Objekt aus dieser Methode zu erzeugen. Dazu müssen wir die Methode angeben, das Ziel der Methode und die an die Methode übergebenen Parameter.

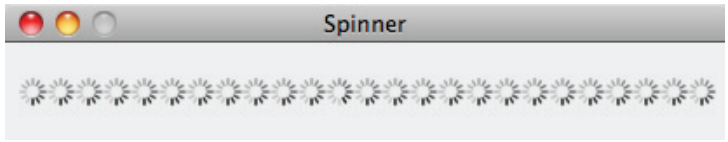
```
NSInvocationOperation *op =
         [[NSInvocationOperation alloc] initWithTarget:self
                                                selector:@selector(spin:)
                                                  object:spinner];
```

Sobald Sie über eine Operation verfügen, können Sie diese in eine Operation-Queue einfügen, die die darin enthaltenen Operationen verwaltet.

Operations/Spinner2/SpinnerAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *arrayOfSpinners = [self arrayOfSpinners];
►   NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    for (NSProgressIndicator *spinner in arrayOfSpinners) {
►       [queue addOperation:[[NSInvocationOperation alloc]
                    initWithTarget:self
                          selector:@selector(spin:)
                            object:spinner]];
    }
}
```

Klicken Sie auf *Build & Run*, und das Ganze sieht ein wenig anders aus:[1] Alle Fortschrittsanzeigen drehen sich gleichzeitig und Sie können das Anwendungsfenster auswählen. Es gibt keinen rotierenden Ball. Das Fenster wird zum aktiven Fenster.



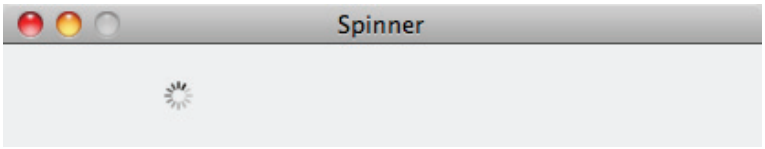
1 Wenn Ihnen das zu schnell geht, können Sie die Wartezeit von einer auf zehn Sekunden erhöhen.

Wie bekommen wir es nun hin, dass sich die Fortschrittsanzeigen nacheinander drehen? Wir können die Zahl der Operationen beschränken, die gleichzeitig ausgeführt werden. Wir können den Wert beispielsweise auf drei setzen und zusehen, wie sich drei Fortschrittsanzeigen gleichzeitig drehen. Als Sonderfall können wir diesen Wert auf eins setzen und aus der Queue eine serielle Queue machen. Die Operationen werden dann nacheinander ausgeführt.

Operations/Spinner3/SpinnerAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *arrayOfSpinners = [self arrayOfSpinners];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
►   [queue setMaxConcurrentOperationCount:1];
    for (NSProgressIndicator *spinner in arrayOfSpinners) {
        [queue addOperation:[[NSInvocationOperation alloc]
                initWithTarget:self
                      selector:@selector(spin:)
                        object:spinner]];
    }
}
```

Nun werden die Fortschrittsanzeigen nacheinander abgearbeitet und die Anwendung reagiert. Keine rotierenden Bälle. Wir können das Fenster auswählen und es wird zum aktiven Fenster.



Ich weiß, dass ich daraus eine große Sache mache. Nun, es ist eine große Sache. Selbst wenn Sie Aufgaben nacheinander erledigen müssen, können Sie die Ansprechbarkeit und Performance Ihrer Anwendung über `NSOperation` erhöhen: Sie verlegen Tasks in Hintergrund-Threads und reservieren den Haupt-Thread für die Interaktion mit dem User.

25.3 Blockoperationen

Wenn Sie für Mac OS X 10.6 (oder höher) entwickeln, können Sie Blöcke anstelle von Methoden verwenden.[2] Sie können ein `NSBlockOperation`-Objekt mit dem folgenden Konstruktor erzeugen:

```
+ (id)blockOperationWithBlock:(void (^)(void))block
```

2 Sie können sich Kapitel 24, *Blöcke*, auf Seite 395 noch einmal ansehen, um sich die Blocksyntax in Erinnerung zu rufen.

Die Signatur bedeutet, dass der Block keine Parameter erwartet und nichts zurückgibt. Das entspricht genau unseren Anforderungen. Legen Sie inline einen Block an, der den gleichen Rumpf hat wie die `spin:`-Methode:

Operations/Spinner4/SpinnerAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *arrayOfSpinners = [self arrayOfSpinners];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [queue setMaxConcurrentOperationCount:1];
    for (NSProgressIndicator *spinner in arrayOfSpinners) {
►       [queue addOperation:[NSBlockOperation blockOperationWithBlock:^{
►               [spinner startAnimation:self];
►               sleep(1);
►               [spinner stopAnimation:self];
►        }]];
    }
}
```

Eliminieren Sie die `spin:`-Methode. Klicken Sie *Build & Run* an, und die Anwendung läuft wie vorhin mit einer Fortschrittsanzeige nach der anderen, und die Anwendung reagiert auch auf Mausklicks.

In diesem Fall können wir auch die `addOperationWithBlock:`-Methode nutzen, die bei Snow Leopard für `NSOperationQueue` hinzugekommen ist. Ersetzen Sie die Zeile

```
[queue addOperation:[NSBlockOperation blockOperationWithBlock:^{
```

durch diese hier:

```
[queue addOperationWithBlock:^{
```

Entfernen Sie auch die schließende eckige Klammer, die auf die geschweifte Klammer folgt, die den Block abschließt:

Operations/Spinner5/SpinnerAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *arrayOfSpinners = [self arrayOfSpinners];
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
     [queue setMaxConcurrentOperationCount:1];
    for (NSProgressIndicator *spinner in arrayOfSpinners) {
►        [queue addOperationWithBlock:^{
            [spinner startAnimation:self];
            sleep(1);
            [spinner stopAnimation:self];
►       }];
    }
}
```

Das war's. Klicken Sie auf *Build & Run*, und alles läuft wie zuvor.

25.4 Interaktion mit der Queue und Operationen

Sie können Informationen aus ihren NSOperations und NSOperationQueues abrufen und Nachrichten senden. Zum Beispiel können Sie ein Array aller aktuellen Operationen in einer Queue abrufen, oder einfach nur die Anzahl der vorhandenen Operationen. Sie können alle Operationen annullieren oder die Queue anhalten und wieder starten. Wir können für eine Operation ermitteln, ob sie annulliert, ausgeführt, abgeschlossen, bereit oder nebenläufig ist. Wir können eine Operation annullieren, bevor sie gestartet wird.

Um das genauer zu untersuchen, fügen Sie drei Aktionen in die Header-Datei ein. Wir verschieben auch die Deklaration der NSOperationQueue in die Header-Datei, damit sie aus mehr als einer Methode heraus aufgerufen werden kann.

Operations/Spinner6/SpinnerAppDelegate.h

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

@interface SpinnerAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
►   NSOperationQueue *queue;
}
@property (assign) IBOutlet NSWindow *window;
► -(IBAction)toggleIsSuspended:(id)sender;
► -(IBAction)cancelAllOperations:(id)sender;
► -(IBAction)queueStatus:(id)sender;
@end
```

Sie müssen die Deklaration der NSOperationQueue aus der applicationDidFinishLaunching:-Methode entfernen. Wo ich gerade dabei bin, ändere ich die maximale Anzahl gleichzeitig ausgeführter Threads nur so zum Spaß auf drei.

Operations/Spinner6/SpinnerAppDelegate.m

```
queue = [[NSOperationQueue alloc] init];
[queue setMaxConcurrentOperationCount:3];
```

Implementieren wir die Aktion, die alle Operationen in der Queue annulliert.

Operations/Spinner6/SpinnerAppDelegate.m

```
-(IBAction)cancelAllOperations:(id)sender {
     [queue cancelAllOperations];
}
```

Wenn Sie eine Queue anweisen, alle Operationen zu annullieren, sendet sie jeder Operation in der Queue die `cancel`-Nachricht. Wurde eine Operation noch nicht gestartet, wird sie annulliert und nicht ausgeführt. Läuft die Operation bereits, liegt es an Ihnen, auf `cancel` zu reagieren (oder auch nicht). Standardmäßig erzwingt `cancel` nicht die Beendigung des Tasks. Sie müssen prüfen, ob `isCancelled YES` oder `NO` zurückgibt, und entsprechend reagieren. Wie das funktioniert, erfahren Sie am Ende des nächsten Abschnitts.

In unserem Beispiel dreht sich jeder laufende `NSProgressIndicator` weiter, wenn alle Operationen annulliert werden. Jede noch nicht laufende Fortschrittsanzeige wird aufgehoben.

Als Nächstes wollen wir es den Benutzern ermöglichen, das Drehen anzuhalten und wieder aufzunehmen:

Operations/Spinner6/SpinnerAppDelegate.m

```
-(IBAction)toggleIsSuspended:(id)sender {
     [queue setSuspended:![queue isSuspended]];
}
```

Wenn Sie der Queue die Nachricht `setSuspended:YES` senden, wird sie angehalten. Wie bei `cancel` laufen die momentan aktiven Fortschrittsanzeigen bis zum Ende durch, nachfolgende Fortschrittsanzeigen pausieren aber so lange, bis Sie der Queue die Nachricht `setSuspended:NO` gesendet haben.

Zum Schluss erzeugen wir ein Array mit den Operationen in der Queue und fragen jede daraufhin ab, ob sie ausgeführt wird oder bereit ist. Ausgeführte Operationen werden als bereit betrachtet.

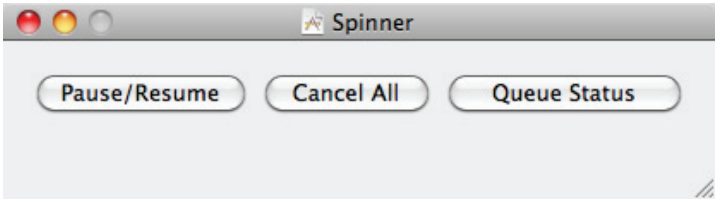
Operations/Spinner6/SpinnerAppDelegate.m

```
-(IBAction)queueStatus:(id)sender {
   NSArray *ops =[queue operations];
   int executing =0;
   int ready=0;
   for (NSOperation *operation in ops) {
       if ([operation isExecuting]) executing++;
       if ([operation isReady]) ready++;
   }
   NSLog(@"Status for %d operations: executing %d and %d are waiting.",
         [queue operationCount], executing, ready -executing);
}
```

Das erzeugt Berichte wie den folgenden in der Konsole. Nachdem neun Operationen abgeschlossen wurden, erhalte ich den folgenden Bericht:

```
Status for 16 operations: executing 3 and 13 are waiting.
```

Wechseln Sie in den Interface Builder und fügen Sie die drei Buttons oben in das Fenster ein:



Verbinden Sie die Aktionen mit diesen Buttons. Speichern Sie Ihre Änderungen und klicken Sie auf *Build & Run*. Die Buttons sollten so funktionieren, wie ich es beschrieben habe.

25.5 Eigene NSOperations

Blockoperationen und der Aufruf von Operationen werden die meisten Ihrer Bedürfnisse abdecken. Wenn nicht, können Sie Ihre eigene Subklasse von `NSOperation` und eigene Operationen entwickeln, die Sie in die Queue einfügen.

Fügen Sie eine neue Datei in Ihr Projekt ein, eine Klasse vom Typ `NSObject` namens `Spinner-Operation`. Wir müssen einige Änderungen an der Header-Datei vornehmen. Die Klasse `SpinnerOperation` muss eine Subklasse der `NSOperation`-Klasse sein. Wir müssen außerdem eine Instanzvariable und eine eigene `init`-Methode für den Spinner deklarieren, wenn wir das `SpinnerOperation`-Objekt initialisieren.

Operations/Spinner7/SpinnerOperation.h

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

@interface SpinnerOperation : NSOperation {
    NSProgressIndicator *spinner;
}
-(id) initWithSpinner:(NSProgressIndicator *) newSpinner;

@end
```

Implementieren Sie die eigene `init`-Methode wie folgt:

Operations/Spinner7/SpinnerOperation.m

```
-(id) initWithSpinner:(NSProgressIndicator *) newSpinner {
    if (self =[super init]) {
        spinner = newSpinner;
    }
    return self;
}
```

Wenn Sie eine Subklasse von `NSOperation` erzeugen und sich keine Gedanken um Nebenläufigkeit machen müssen, tragen Sie die Arbeit, die von dieser Operation erledigt werden muss, in die `main`-Methode ein.[3]

Operations/Spinner7/SpinnerOperation.m

```
-(void) main {
     [spinner startAnimation:self];
     sleep(4);
     [spinner stopAnimation:self];
}
```

Die restlichen Änderungen stehen in der `SpinnerAppDelegate.m`-Datei. Nachdem Sie einen `import` der neuen Header-Datei eingefügt haben, ändern Sie zuerst die `arrayOfSpinners`-Methode in diese `arrayOfSpinnerOperations`-Methode ab:

Operations/Spinner7/SpinnerAppDelegate.m

```
► -(NSArray *) arrayOfSpinnerOperations {
    NSMutableArray *array = [[NSMutableArray alloc] initWithCapacity:25];
    for (int i =0;i < 25; i++){
        NSProgressIndicator *spinner = [[NSProgressIndicator alloc]
            initWithFrame: NSMakeRect(16 * i + 8, 20, 16, 16)];
         [spinner setStyle:NSProgressIndicatorSpinningStyle];
         [spinner setControlSize:NSSmallControlSize];
         [spinner setDisplayedWhenStopped:NO];
         [window.contentView addSubview:spinner];
►        SpinnerOperation *op = [[SpinnerOperation alloc]
►                                initWithSpinner:spinner];
►        [array addObject:op];
    }
    return array;
}
```

Die beiden Methoden sind sich sehr ähnlich. Neben der Namensänderung übergeben wir die von uns erzeugte und konfigurierte Fortschrittsanzeige an die `SpinnerOperation-initWithSpinner:`-Methode und fügen das `SpinnerOperation`-Objekt in das Array ein statt in einen `NSProgressIndicator`.

Die `applicationDidFinishLaunching:`-Methode ist nun wesentlich klarer. Sie konzentriert sich auf die Operation-Queue und nicht auf die Arbeit, die die Operationen durchführen.

---

3 Die `NSOperation`-Dokumentation zeigt, dass sehr viel zusätzliche Arbeit nötig ist, wenn Ihre Operationen nebenläufig ausgeführt werden sollen. Das ist hier nicht der Fall und wird niemals der Fall sein, wenn Sie Ihre Operationen mit Operation-Queues nutzen. Die `NSOperation`-Dokumentation macht auch deutlich, dass Sie Ihre Operationen nicht nebenläufig auslegen müssen, wenn Sie eine Operation-Queue verwenden.

Operations/Spinner7/SpinnerAppDelegate.m

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
    NSArray *arrayOfSpinnerOperations = [self arrayOfSpinnerOperations];
    queue = [[NSOperationQueue alloc] init];
    [queue setMaxConcurrentOperationCount:1];
    for (SpinnerOperation *spinnerOp in arrayOfSpinnerOperations) {
        [queue addOperation:spinnerOp];
    }
}
```

Ich möchte eine weitere Änderung vornehmen, damit die Anwendung noch besser reagiert. In unserem Fall ist das trivial – der Thread pausiert für eine bestimmte Zeit. Wir können diese Zeitspanne in kleinere Stücke aufteilen und den Status des `cancelled`-Flags prüfen.

Operations/Spinner8/SpinnerOperation.m

```
-(void) main {
    [spinner startAnimation:self];
    for (int i = 0; i<4; i++){
        if ([self isCancelled]) break;
        sleep(1);
     }
     [spinner stopAnimation:self];
}
```

Klicken Sie auf *Build & Run*. Nun können Sie alle Operationen mitten in einer laufenden Operation annullieren.

25.6 Von Operation-Queues zu Dispatch-Queues

Nebenläufigkeit ist eine schwierige Sache. Doch wie Sie in diesem (zugegebenermaßen etwas gekünstelten) Beispiel gesehen haben, ist der besonnene Einsatz von Nebenläufigkeit das Mittel, um dem Benutzer das Gefühl zu geben, dass Computer und Anwendung schnell reagieren.

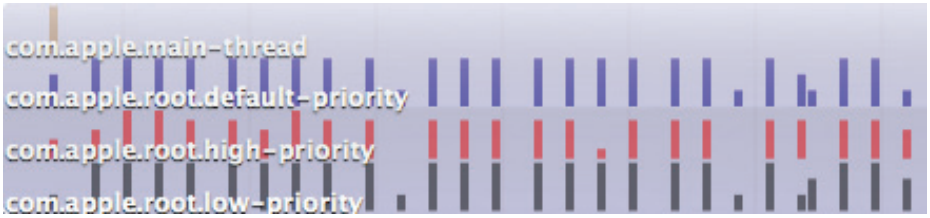
Die Technik der Operation-Queues ist wesentlich einfacher als die Technik der Dispatch-Queues, die ich Ihnen im nächsten Kapitel vorstellen werde. Operation-Queues sind gradlinig und recht einfach zu benutzen. Dispatch-Queues sind eine auf niedriger Ebene angesiedelte, C-basierte API, die mehr Details darüber wissen muss, was gerade passiert. Sie sollten immer mit den höher angesiedelten Abstraktionen anfangen, in diesem Fall also mit Operation-Queues, bevor Sie die C-APIs für Dispatch-Queues nutzen.

Wenn Sie für Snow Leopard (oder höher) entwickeln: Die Klassen, mit denen Sie in diesem Kapitel gearbeitet haben, wurden so angepasst, dass sie mit Grand Central Dispatch (dem Marketingbegriff für die Infrastruktur, die Dispatch-Queues verwaltet) zusammenarbeiten.

Sie glauben mit nicht? Ich zeige es Ihnen.

Öffnen Sie Xcode und kompilieren Sie die gerade entwickelte Spinner-Anwendung. Sobald die Kompilierung erfolgreich abgeschlossen ist, wählen Sie `Run > Run with Performance Tool > Multicore`. Das startet Instruments mit zwei Instrumenten: Das Thread States-Instrument befindet sich oben und das Dispatch-Instrument unten. Die Spinner-App läuft, während Instruments seine Aktivität aufzeichnet.

Hier die grafische Zusammenfassung des Dispatch-Instruments:



Das Ganze ist etwas schwer zu lesen, aber der Text auf der linken Seite identifiziert die vier globalen Dispatch-Queues, die Sie im nächsten Kapitel explizit nutzen werden. Der Haupt-Thread steht oben. Dort geschehen das, was in die Haupt-Queue eingestellt wurde. In der Haupt-Queue und im damit verbundenen Haupt-Thread erfolgt das ganze UI-Handling.

Sie können erkennen, dass die drei unteren Queues recht viel beschäftigt werden, damit der Haupt-Thread für Benutzereingaben zur Verfügung steht. Die drei unteren Queues sind globale Queues, die Arbeiten für drei unterschiedliche Prioritätsstufen akzeptieren. Von oben nach unten sind das die Standardpriorität, hohe Priorität und niedrige Priorität. Sie können auch erkennen, dass die Last relativ gleichmäßig verteilt wird.

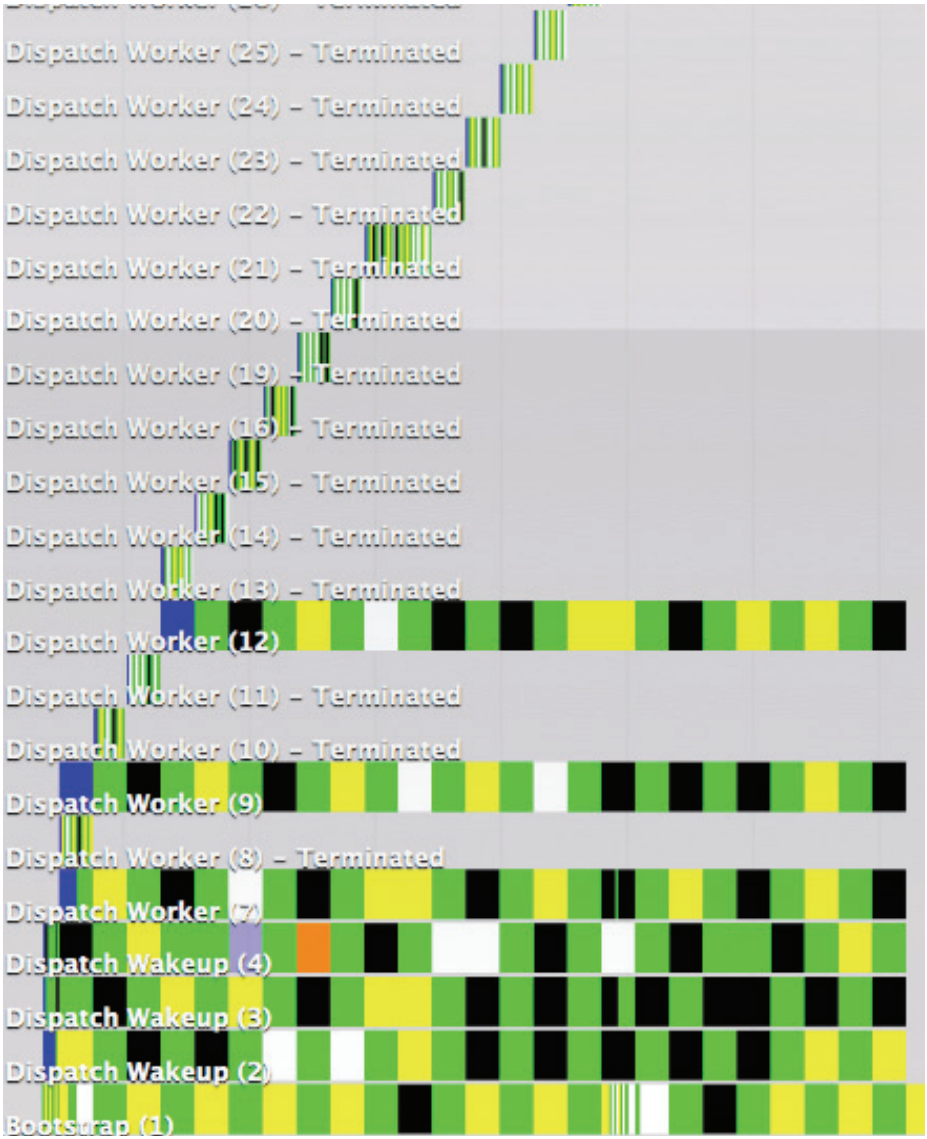
Vergleichen Sie das mit einem Lauf des Spinners ohne Queues.

Sie können erkennen, dass das Ganze fast genauso abläuft wie in der nebenläufigen Version. Doch nachdem die Arbeit verteilt wurde, passiert nichts mehr, bis nach 25 Sekunden der letzte Spinner fertig ist. Sie können sich in Instruments umsehen, wenn Sie genauer wissen wollen, was hier vor sich geht. Doch selbst ein kurzer Blick auf die beiden folgenden Berichte zeigt, dass hier etwas anders ist.

Werfen Sie einen Blick auf die Berichte des Thread States-Instruments, und Sie erkennen noch mehr Details. Wir erzeugen und nutzen Threads nicht explizit, sondern das wird hinter den Kulissen für uns erledigt. Hier der erste Bericht der ursprünglichen (nicht reagierenden) Version von Spinner:

Vergleichen Sie das mit dem unteren Teil des Berichts für die nebenläufige Version. Das diagonale Muster von Threads läuft weiter bis nach rechts.

Auf der folgenden Abbildung können Sie sehen, dass der Inhalt von Dispatch Worker 7 in der ersten Version über alle Threads im oberen Teil des Bildes der zweiten Variante verteilt wird. Wieder können Sie erkennen (ohne zu sehr auf die Details einzugehen), dass die nebenläufige Version performanter ist und besser reagiert.



Im nächsten Kapitel wollen wir in die Details eintauchen und explizit Grand Central Dispatch verwenden. Doch die Chancen stehen gut, dass Sie nur Operation-Queues verwenden müssen, um das gewünschte Verhalten zu erreichen.

Kapitel 26

Dispatch-Queues

Computer werden mit immer mehr Kernen ausgeliefert, und eine Möglichkeit, um Ihre Anwendung schneller zu machen, besteht darin, diese Kerne auszulasten. Im vorigen Kapitel haben Sie die einfachste Möglichkeit kennengelernt, Ihre Anwendung die Vorteile der Nebenläufigkeit nutzen zu lassen: Operation-Queues.

Manchmal benötigt man aber größere Kontrolle oder muss mit Aktivitäten auf Systemebene interagieren. Wenn Sie für das iPhone entwickeln oder für Leopard (und älter), haben Sie Pech gehabt. Sie müssen mit Threads arbeiten. Die Arbeit mit Threads ist schwierig. Man macht sehr leicht Fehler und kann die Leistung der Anwendung sogar verschlechtern.

Mit Snow Leopard hat Apple Grand Central Dispatch (GCD) eingeführt. Es überträgt die Verantwortung für die Verwaltung der Tasks innerhalb der Threads auf das Betriebssystem. Sie fügen Tasks in Dispatch-Queues ein, und GCD kümmert sich darum, sie in den verschiedenen Queues auf die Prozessoren zu verteilen.

In diesem Kapitel werden Sie erfahren, wie und wann man die verschiedenen Arten von Dispatch-Queues nutzt. Wir kratzen nur an der Oberfläche eines komplexen und schwierigen Themas, aber Sie bekommen eine Vorstellung davon, was möglich ist und wie man Dispatch-Queues in eine Anwendung integriert.

26.1 Wann man Dispatch-Queues nutzt

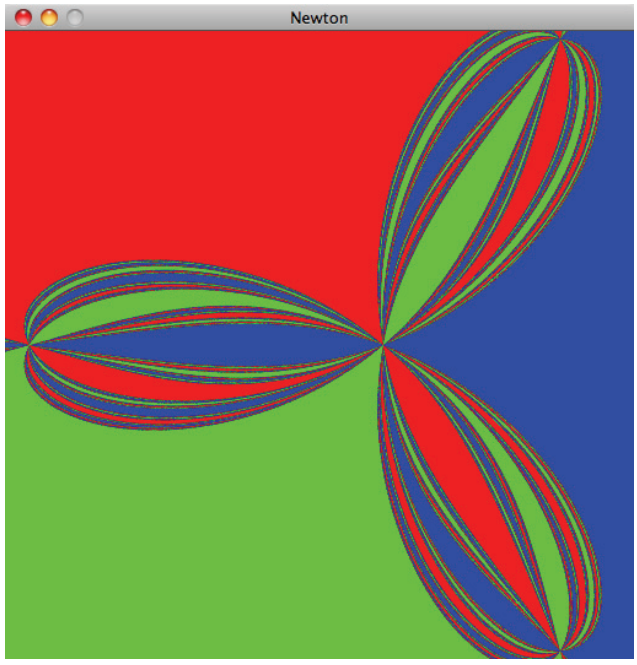
Stellen Sie sich vor, Sie haben ein Programm entwickelt, das verschiedene Websites nach den günstigsten Flügen zu einem bestimmten Ort an einem bestimmten Tag durchsucht. Das Programm würde den Benutzer

fragen, wann die Reise beginnt, wann sie endet und wohin es geht. Dafür würden Sie nicht eine Site nach der anderen durchsuchen, denn das könnte der Benutzer ja auch selbst machen. Sie würden alle Anfragen asynchron senden und die Ergebnisse sammeln und ausgeben.

Das ist ein Beispiel für die Art von Problem, für die Dispatch-Queues gut geeignet sind. Sie haben eine Reihe von Aufgaben, die gleichzeitig ausgeführt werden können, und jede Aufgabe dauert so lange, dass Sie das Programm nicht auf ihren Abschluss warten lassen wollen.

Sobald man sich nach Tasks umsieht, die von Dispatch-Queues profitieren, findet man sie überall. Stellen Sie sich eine Reihe von Bildern vor, auf die Sie einen Filter anwenden wollen: Wenn Ihnen mehrere Kerne zur Verfügung stehen, können Sie den Filter zur selben Zeit auf verschiedene Bilder anwenden und die Aufgabe schneller erledigen. Für einige Filter kann man die Ausführung sogar noch beschleunigen, indem man das Bild in kleinere Teile zerlegt, den Filter auf die einzelnen Teile anwendet und das Bild zum Schluss wieder zusammensetzt.

Das Beispiel in diesem Kapitel konzentriert sich auf eine lange Folge von Berechnungen, die wir auf Hunderte von Tausenden quadratischer Bereiche eines Rasters anwenden, um die Farbe zu ermitteln, mit der wir dieses Quadrat füllen wollen.



Die Details des gezeichneten Fraktals sind nicht weiter wichtig. Was es zu einem guten Kandidaten für Dispatch-Queues macht, ist die Tatsache, dass die Berechnung für jedes dieser über 250.000 Quadrate unabhängig von den anderen erfolgen kann und dass die Berechnungen nicht trivial sind. In unserem Beispiel sind Tausende von Berechnungen notwendig, um die Farbe eines jeden Quadrats zu bestimmen. Wir haben also viel Arbeit zu erledigen, die man einfach parallelisieren kann.

26.2 Eine kurze Queue-Übersicht

Es gibt drei grundlegende Arten von Dispatch-Queues:

- nebenläufige (concurrent) Queues,
- die Haupt-Dispatch-Queue und
- private Dispatch-Queues.

Wir haben uns nebenläufige Queues und die Haupt-Queue am Ende des vorigen Kapitels schon kurz angesehen.

Globale nebenläufige Queues verwenden Sie für parallelisierbare Operationen. Es gibt drei verschiedene nebenläufige Queues, nämlich mit niedriger, mittlerer und hoher Priorität. Zwar stehen die Tasks in jeder Queue für die parallele Ausführung zur Verfügung, aber grundsätzlich sind alle Dispatch-Queues FIFO-Queues (first in, first out), die Tasks werden also in der Reihenfolge gestartet, in der sie in die Queue eingetragen wurden. Tasks mit hoher Priorität werden vor Tasks mit normaler Priorität ausgeführt, und so weiter.

Manchmal möchte man eine Reihe von Tasks in einer bestimmten Reihenfolge ausführen. In diesem Fall übergeben Sie die Tasks an eine globale serielle Queue. Das werden Sie häufig für GUI-Operationen nutzen, oder zur Synchronisation mit dem Event-Loop. Sie können es auch nutzen, um den Zugriff auf eine gemeinsam genutzte Ressource innerhalb der Anwendung zu synchronisieren.

Wenn Sie Tasks nur synchronisieren wollen, können Sie eine serielle Queue anlegen, die sicherstellt, dass die an die Queue übergebenen Tasks nacheinander ausgeführt werden. Sie können die Haupt-Queue zur Synchronisation von Tasks nutzen, da auch sie eine serielle Queue ist, doch das Ziel besteht darin, die Performance und das Ansprechverhalten zu verbessern, indem man Tasks aus der Haupt-Queue verschiebt.

Sie können sogar einen Block oder eine Funktion an eine Queue übergeben. In diesem Buch wollen wir mit Blöcken arbeiten. Sie können Blöcke synchron oder asynchron in die Queues einfügen. In den meisten Fällen werden Sie die `dispatch_asynch()`-Funktion bevorzugen, um einen Block in eine Queue einzufügen und gleich wieder zurückzukehren. Richtig – wir müssen die C-Syntax nutzen, da wir mit guten alten C-Funktionen und Variablen arbeiten.

Wenn Sie warten müssen, bis ein Task ausgeführt wurde, bevor Sie mit dem Programm fortfahren können, fügen Sie den Block mit `dispatch_synch()` in die Queue ein. Es gibt weitere Techniken, etwa Gruppen, mit denen Sie koordinieren können, welche Arbeit wann erledigt wird. Wir werden sie in diesem Buch aber nicht behandeln. Sobald Sie wissen, wie Sie die elementaren Optionen nutzen, sollten Sie Apples *Concurrency Programming Guide* [App09c] lesen.

26.3 Unser Fraktal zeichnen

Beginnen Sie mit dem Newton-Projekt, das Sie im Verzeichnis `Dispatch/Newton1` im Codedownload finden. Neben dem Application-Delegate enthält es noch zwei Klassen. Die `Grid`-Klasse ist ein `NSView`, den wir zur Darstellung der Ergebnisse nutzen. `Grid` erzeugt eine Instanz der `Tile`-Klasse, die ein Rechteck repräsentiert, das wir einfärben wollen. Das `Tile`-Objekt berechnet seine Farbe und bringt vier Unterrechtecke hervor, deren Farbe berechnet werden muss. Wir wiederholen diesen Zyklus des Einfärbens eines Rechtecks und der Aufteilung in vier Unterrechtecke, bis wir unser 512 x 512-Grid mit 262.144 Quadraten gefüllt haben.

Die `Grid`-Klasse besitzt eine Eigenschaft vom Typ `NSMutableArray` namens `tiles`, die die Rechtecke und ihre Farben enthält. Sie können im folgenden Code sehen, dass wir das Array in der `initWithFrame:`-Methode initialisieren. Für die Zeichnung kopieren wir das Array, gehen die Tiles innerhalb des zu aktualisierenden Rechtecks durch und zeichnen diejenigen, die neu gezeichnet werden müssen. Die `startTiling`-Methode erzeugt das erste `Tile` und stößt die Aktion an.

Dispatch/Newton1/Grid.m

```
#import "Grid.h"
#import "Tile.h"

@implementation Grid
@synthesize tiles;
```

```
- (id)initWithFrame:(NSRect)frame {
    if (self=[super initWithFrame:frame]) {
        self.tiles = [[NSMutableArray alloc] initWithCapacity:1000];
    }
    return self;
}
-(void)startTiling {
     [[[Tile alloc] initWithFrame:self.frame view:self] cycle];
}
-(void)drawRect:(NSRect)dirtyRect {
    for (Tile *tile in [NSArray arrayWithArray:tiles]){
        if(NSContainsRect(dirtyRect, tile.frame)) {
             [tile.color set];
             [NSBezierPath fillRect:tile.frame];
        }
    }//NSLog( @"here");
}
@end
```

Bei der Einführung in Dispatch-Queues werden wir diese Dateien größtenteils nicht anrühren. Im nächsten Abschnitt zeige ich Ihnen die Bereiche der `Tile`-Klasse, die wir mit Queues optimieren.

26.4 Ohne Dispatch-Queues arbeiten

Bei dieser Anwendung wird ein Großteil der Arbeit in der `Tile`-Klasse erledigt. Auf der obersten Ebene sieht das Leben eines `Tile`-Objekts wie folgt aus:

Dispatch/Newton1/Tile.m

```
-(void) cycle {
    [self calculateColor];
        [self.grid.tiles addObject:self];
    [self.grid setNeedsDisplayInRect:self.frame];
    if (self.frame.size.width > 2) {
        [self split];
    }
}
```

Wir rufen die `calculateColor`-Methode auf, die 2.000-mal Newtons Methode durchläuft, und setzen die Farbe des aktuellen `Tile`-Objekts. Der Algorithmus ließe sich zwar noch verbessern, aber in diesem Kapitel wollen wir uns ansehen, wie die Performance und das Ansprechverhalten mithilfe von Dispatch-Queues verbessert werden kann.

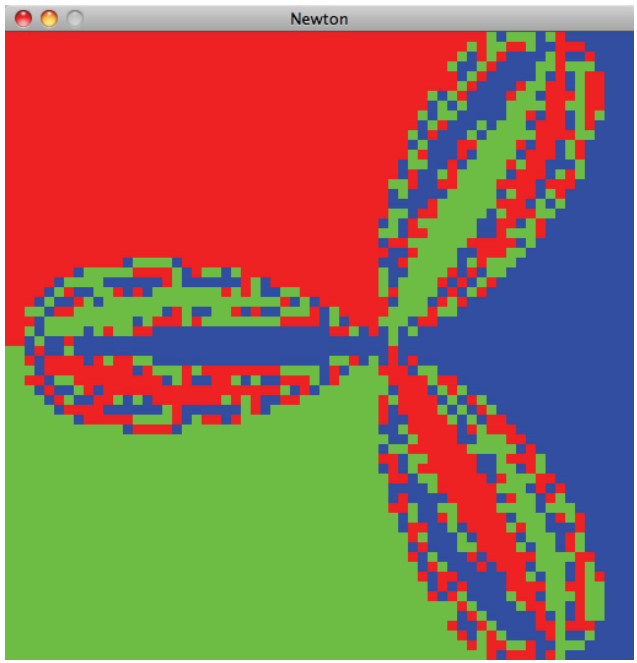
Der Rest der `cycle`-Methode fügt die aktuelle „Kachel“ in das `tiles`-Array des Grids ein und überprüft, ob eine weitere Unterteilung notwendig ist. Die `split`-Methode erzeugt vier gleich große „Unterkacheln", die die aktuelle Kachel abdecken.

Dispatch/Newton1/Tile.m

```
-(void) split {
    CGFloat size = (self.frame.size.width)/2;
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y;
    [self tileWithX:x Y:y size:size];
    [self tileWithX:x Y:y+size size:size];
    [self tileWithX:x+size Y:y size:size];
    [self tileWithX:x+size Y:y+size size:size];
    [self.grid.tiles removeObject:self];
}
```

Ich habe mich dafür entschieden, mit einem Quadrat mit einer Seitenlänge von 512 Punkten zu beginnen. Ich unterteile die Quadrate jedesmal und bestimme ihre Farbe. Sobald wir mit Threads arbeiten, können wir die minimale Größe der Quadrate auf 1 festlegen, aber im Moment höre ich auf, sobald das Quadrat 8 Punkte breit ist.

Führen Sie das Programm über *Build & Run* mit verschiedenen Werten für die minimale Breite aus. Auf meinem Laptop sehe ich das Ergebnis fast sofort, wenn die minimale Breite bei 8 liegt. Hier das fertige Bild bei einer minimalen Kachelbreite von 8:



Bei 4 gibt es eine kurze Verzögerung, bis das Ergebnis erscheint. Setze ich den Wert auf 2, gibt es eine längere Pause und ich sehe nichts außer dem berühmten rotierenden Ball. Fast 20 Sekunden lang sehe ich nur einen weißen Bildschirm, und dann erscheint sofort das fertige Bild.

Geben Sie in der `drawRect:`-Methode in `Grid.m` eine Meldung über die Konsole aus, damit Sie sehen, dass diese Methode nur einmal aufgerufen wird.

Stellen Sie sich vor, dass Sie die `drawRect:`-Methode direkt aufrufen, statt `setNeedsDisplayInRect:` zu verwenden. (Tun Sie das bloß nicht, stellen Sie es sich nur vor!) In diesem Fall wird `drawRect:` bei jedem Durchlaufen der Schleife aufgerufen, doch es wird nichts ausgegeben, solange die Berechnung nicht abgeschlossen ist. Zusätzlich erhöht der direkte Aufruf die Laufzeit der Anwendung. Lassen Sie den Aufruf bei `setNeedsDisplayInRect:` und nicht bei `drawRect:`.

Nun wollen wir ein wenig Multithreading nutzen.

26.5 Die Haupt-Queue

Rufen Sie sich ins Gedächtnis, dass Ihnen mehrere globale Queues zur Verfügung stehen und dass die mit dem Haupt-Thread verknüpfte Queue eine serielle Queue ist, die *Haupt-Queue* (main queue) genannt wird. Ereignisse und Änderungen an der GUI erfolgen im Haupt-Thread. Sobald Sie damit anfangen, asynchrone Threads zu benutzen, um die Performance Ihrer Anwendung zu erhöhen, werden Sie Updates in den Haupt-Thread schieben, damit die Benutzer nicht den rotierenden Ball anstarren müssen. Wir werden ein paar Abschnitte brauchen, um das richtig hinzubekommen.

Sie legen die Haupt-Queue und die globalen Queues nicht selbst an. Sie rufen eine Funktion auf, die Ihnen eine Referenz auf die Queue zurückliefert, die Sie dann anfordern. Im Fall der Haupt-Queue benutzen Sie die Funktion `dispatch_get_main_queue( )`, um an diese Referenz zu gelangen. Ein Handle auf die Haupt-Queue rufen und speichern Sie wie folgt ab. Beachten Sie, dass der Typ `dispatch_queue_t` lautet.

```
dispatch_queue_t main = dispatch_get_main_queue();
```

Wir werden Blöcke entweder über `dispatch_async()` oder über `dispatch_sync()` an Queues übergeben. Meistens werden wir `dispatch_async()` verwenden, damit wir nicht darauf warten müssen, dass der Task abge-

arbeitet wird. Beide Methoden verlangen zwei Argumente. Das erste ist die Queue und das zweite der Block, der den auszuführenden Task beschreibt.

```
dispatch_async(main,^{
    //Auszuführender Code
});
```

Wir ändern die `cycle`-Methode so ab, dass die Requests zum Einfügen einer Kachel in das `Grid`-Array sowie zu Darstellung des Ergebnisses an den Haupt-Thread übergeben werden.

Dispatch/Newton2/Tile.m

```
-(void) cycle {
    [self calculateColor];
►   dispatch_async(dispatch_get_main_queue(), ^{
        [self.grid.tiles addObject:self];
        [self.grid setNeedsDisplayInRect:self.frame];
►   });
    if (self.frame.size.width > 2 ) {
        [self split];
    }
}
```

Auf meinem Laptop ist die Performance besser als vorher. Die Zeit bei einer minimalen Breite von 2 hat sich von 20 auf unter fünf Sekunden verkürzt. Die Zeit für eine Breite von 1 liegt nun bei etwa 20 Sekunden. Als Nächstes wollen wir die Berechnungen in Queues packen, die nicht im Haupt-Thread ausgeführt werden.

26.6 Globale nebenläufige (concurrent) Queues

Die nächste Änderung ist verblüffend. Die Performance verbessert sich ein wenig, doch der Hauptunterschied besteht darin, dass das Zeichnen direkt beginnt und das Bild bis zur Fertigstellung aktualisiert wird.

Wenn der Haupt-GUI-Thread nicht mitintegriert werden muss, kann man eine Operation gefahrlos an einen anderen Thread übergeben. Wenn es geht, sollten Sie die globale nebenläufige Queue nutzen. Sie erhalten ein Handle auf diese Queue über den Aufruf der Funktion `dispatch_get_global_queue()`. Das erste Argument gibt die Priorität der Queue an. Sie können zwischen `DISPATCH_QUEUE_PRIORITY_DEFAULT`, `DISPATCH_QUEUE_PRIORITY_LOW` und `DISPATCH_QUEUE_PRIORITY_HIGH` wählen. Die Dokumentation erklärt, dass das zweite Argument auf `0` gesetzt werden muss, da dieser Slot für zukünftige Erweiterungen reserviert ist.

Schon bald werden wir viele dieser nebenläufigen Queues nutzen. Im Moment wollen wir nur die Aufrufe für das Anstoßen der vier Unterkacheln innerhalb der `split`-Methode an eine nebenläufige Queue übergeben. Wir verwenden dabei die Standardpriorität.

Dispatch/Newton3/Tile.m

```
-(void) split {
    CGFloat size = (self.frame.size.width)/2;
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y;
►   dispatch_async(
►     dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        [self tileWithX:x Y:y size:size];
        [self tileWithX:x Y:y+size size:size];
        [self tileWithX:x+size Y:y size:size];
        [self tileWithX:x+size Y:y+size size:size];
►    });
     [self.grid.tiles removeObject:self];
}
```

Das Ergebnis ist wesentlich zufriedenstellender. Ein paar von den großen Quadraten erscheinen sehr schnell, und dann erscheint das Fraktal animiert, während es verfeinert wird. Probieren Sie das mit verschiedenen Werten für die minimale Breite der Kacheln aus. Das Programm ist ein wenig schneller als zuvor und, fast noch wichtiger: Es macht Spaß, zuzusehen.

26.7 Synchronisation über die Haupt-Queue

Wir haben die Anwendung beschleunigt; sie sieht besser aus und fühlt sich auch besser an. Aber wir haben ein Problem: Da es sich um eine nebenläufige Anwendung handelt, sehen Sie in der Konsole immer die folgende Meldung, wenn Sie das Programm ausführen. Diese Warnung erscheint so gut wie immer, wenn die minimale Größe klein genug ist:

```
*** -[NSCFArray initWithObjects:count:]:
                    attempt to insert nil object at objects[5733]
```

Das Problem besteht darin, dass wir Objekte in die `tiles`-Eigenschaft einfügen und aus ihr entfernen, die ein Array ist, das unserem `Grid`-Objekt gehört. Wir fügen die Objekte aus der Haupt-Queue (einer globalen seriellen Queue) in das Array ein, entfernen sie aber nicht aus dieser Queue heraus. Das Programm könnte also prüfen, ob die Kachel neu gezeichnet werden muss, nachdem es bereits aus dem Array entfernt wurde. Wir können das Problem beheben, indem wir den Aufruf zum Entfernen von Objekten aus dem `tile`-Array in die Haupt-Queue verschieben.

Dispatch/Newton4/Tile.m

```
-(void) split {
    CGFloat size = (self.frame.size.width)/2;
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y;
►   dispatch_async(
►     dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
        [self tileWithX:x Y:y size:size];
        [self tileWithX:x Y:y+size size:size];
        [self tileWithX:x+size Y:y size:size];
        [self tileWithX:x+size Y:y+size size:size];
    });
►   dispatch_async(dispatch_get_main_queue(), ^{
    [self.grid.tiles removeObject:self];
    });
}
```

Die Haupt-Queue ist eine serielle Queue: Wenn wir auf das Array nur über die Haupt-Queue zugreifen, kommt es auch nicht zu einem Streit um diese Ressource. In der Haupt-Queue werden ebenfalls Ereignisse verarbeitet, hier können Sie also auch das notwendige UI-Handling erledigen.

Sie werden bemerkt haben, das die Animation anders aussieht. Mir hat es vorher besser gefallen, als zuerst der gesamte View eingefärbt und dann bei jeder weiteren Berechnung verfeinert wurde. Jetzt beginnt die „Animation“ in der unteren linken Ecke und arbeitet sich nach rechts oben hoch.

26.8 Private Dispatch-Queues

Die Haupt-Queue zur Synchronisation des Zugriffs auf das `tiles`-Array zu nutzen, ist durchaus sinnvoll, da wir sie für das Zeichnen verwenden, das im Haupt-Thread erfolgt. Außerdem synchronisieren wir die Aufrufe zwischen Tausenden von `Tile`-Objekten.

Manchmal wollen wir eine kleine Gruppe von Tasks über eine kurze Zeitspanne oder in einem kurzen Codeabschnitt zusammenfassen. Das ist mit seriellen Queues möglich, die Sie selbst erzeugen und wieder freigeben. Die seriellen Queues sind nicht global, Sie erzeugen sie also selbst. Der Aufruf sieht so aus:

```
dispatch_queue_t myQ = dispatch_queue_create("com.pragprog.myQ",NULL);
```

Der erste Parameter ist ein Label für Ihre Queue, die beim Debugging genutzt wird. Nutzen Sie ein umgekehrtes DNS-Schema und geben Sie Ihrer Queue einen eindeutigen Namen, damit Sie herausfinden können,

wo das Ganze schiefläuft. Der zweite Parameter wird momentan nicht genutzt, und die Dokumentation sagt, dass Sie einfach `NULL` übergeben sollen.

Es handelt sich um eine serielle Queue, jeder von uns in die Queue eingefügte Task wird also ausgeführt, bevor der nächste Task beginnt. In unserem Beispiel wollen wir zwei Tasks in die Queue einfügen. Tatsächlich handelt es sich bei jedem Task um einen Queue-Dispatch. Im ersten stoßen wir die vier Unterkacheln an und im zweiten entfernen wie die Parent-Kachel aus dem `tiles`-Array.

Dispatch/Newton5/Tile.m

```
-(void) split {
    CGFloat size = (self.frame.size.width)/2;
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y;
►   dispatch_queue_t myQ =
►                       dispatch_queue_create("com.pragprog.myQ", NULL);
►   dispatch_async(myQ, ^{
        dispatch_async(
            dispatch_get_global_queue(
                                DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
             [self tileWithX:x Y:y size:size];
             [self tileWithX:x Y:y+size size:size];
             [self tileWithX:x+size Y:y size:size];
             [self tileWithX:x+size Y:y+size size:size];
►       });
►   });
►   dispatch_async(myQ, ^{
        dispatch_async(dispatch_get_main_queue(), ^{
            [self.grid.tiles removeObject:self];
        });
►   });
►   dispatch_release(myQ);
}
```

Diese privaten Dispatch-Queues arbeiten mit Referenzzählern und werden momentan durch die automatische Garbage Collection nicht bereinigt. Wie Sie in der letzten hervorgehobenen Zeile sehen können, geben wir die Queue explizit frei:

```
dispatch_release(myQ)
```

Müssten Sie eine dieser Queues weiterreichen, dann müsste jedes Objekt, dem die Queue gehört, diese explizit festhalten und später wieder freigeben. Wenn Sie eine serielle Queue über `dispatch_queue_create()` erzeugen, ist der Referenzzähler für diese Queue 1. Aus diesem Grund müssen wir sie manuell freigeben, sobald wir sie nicht mehr benötigen.

Klicken Sie *Build & Run* an, und alles funktioniert ausgezeichnet. Allerdings gibt es hier noch jede Menge Ballast. Sie erzeugen eine ganze Menge von Dispatch-Queue-Objekten und geben sie wieder frei. Sie können sich das ansehen, wenn Sie diese Version des Projekts mit dem Dispatch-Instrument von Instruments ansehen. Sie können erkennen, dass Unmengen von Queues für eine sehr kurze Zeitspanne erzeugt und verwendet werden.

Wenn Sie eine Queue erzeugen und nutzen, wollen Sie im Allgemeinen sicherstellen, dass es genug Arbeit zu erledigen gibt, um ihren Einsatz zu rechtfertigen. Leider können Sie diesen Overhead manchmal nicht vermeiden. Nun wollen wir uns eine Situation ansehen, in der wir weder die Haupt-Queue noch die privaten Dispatch-Queues verwenden müssen.

26.9 Synchrone Tasks

Alles läuft so viel schneller und besser, dass Sie versucht sein könnten, Dispatch-Queues überall zu verwenden. Zum Beispiel können wir die Berechnung der Farbe einer Kachel in eine nebenläufige Queue verschieben:

Dispatch/Newton6/Tile.m

```
-(void) cycle {
►    dispatch_async(
         dispatch_get_global_queue(
                         DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
              [self calculateColor];
►        });
     dispatch_async(dispatch_get_main_queue(), ^{
          [self.grid.tiles addObject:self];
          [self.grid setNeedsDisplayInRect:self.frame];
     });
     if (self.frame.size.width > 1 ) {
      [self split];
     }
}
```

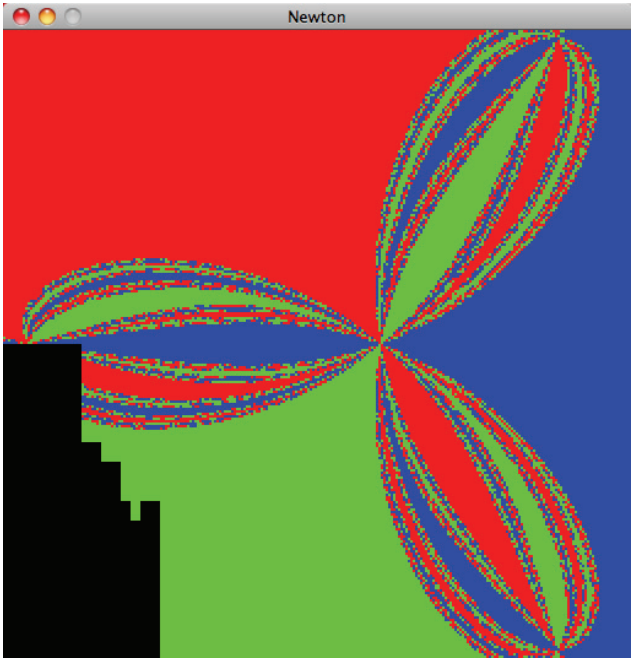
Das Problem bei diesem Ansatz lässt sich besser erkennen, wenn wir die serielle Queue herausnehmen, die wir im vorigen Abschnitt eingefügt haben:

Dispatch/Newton6/Tile.m

```
-(void) split {
    CGFloat size = (self.frame.size.width)/2;
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y;
    dispatch_async(
```

```
        dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0),^{
            [self tileWithX:x Y:y size:size];
            [self tileWithX:x Y:y+size size:size];
            [self tileWithX:x+size Y:y size:size];
            [self tileWithX:x+size Y:y+size size:size];
        });
}
```

Klicken Sie *Build & Run* an. Ich sehe Farbkleckse auf einem schwarzen Hintergrund. Der Code zum Zeichnen der Rechtecke wird aufgerufen, bevor die Farbe berechnet wurde. Sie sehen variierende Mengen von Schwarz, wenn Sie die Anzahl der Iterationen in Newtons Methode von 2000 auf 200 oder 20 ändern.



Wir haben es mit einer sogenannten „Race Condition“ zu tun: Es gibt keine Garantie dafür, dass `calculateColor` ausgeführt wird, bevor `self` in der Haupt-Queue gezeichnet wird. Das verdeutlicht das Problem, das wir im vorigen Abschnitt mithilfe serieller Dispatch-Queues vermieden haben. Bei diesem Code wird die Berechnung der Farbe einer Kachel mit `dispatch_async()` an einen globalen nebenläufigen Thread übergeben. Wir müssen nicht darauf warten, dass die Berechnung der Farbe abgeschlossen ist, bevor wir einen asynchronen Request zum Zeichnen der Kachel an die Haupt-Queue senden.

Lassen Sie uns eine kleine Änderung am Code vornehmen. Ändern Sie den ersten Aufruf Funktion `dispatch_async()` in `cycle` in `dispatch_sync()` ab.

Dispatch/Newton7/Tile.m

```
-(void) cycle {
►   dispatch_sync(
       dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
              [self calculateColor];
        });
   dispatch_async(dispatch_get_main_queue(), ^{
         [self.grid.tiles addObject:self];
         [self.grid setNeedsDisplayInRect:self.frame];
    });
    if (self.frame.size.width > 1 ) {
        [self split];
    }
}
```

Nun wir die Berechnung der Kachelfarbe abgeschlossen, bevor wir das Zeichnen der Kachel an den Haupt-Thread übergeben. Diese Version funktioniert genau wie die Version mit den seriellen Threads, aber ohne den Overhead der Erzeugung dieser Queues.

In dieser kurzen Einführung in Dispatch-Queues haben wir uns drei grundlegende Arten von Queues angesehen. Wir haben auch gesehen, wie man diesen Queues über die Funktionen `dispatch_async()` und `dispatch_sync( )` Tasks übergibt. Wenn unser Beispielprogramm nicht bloß Mittel zum Zweck wäre, gäbe es noch eine ganze Menge zu tun. Wir könnten die Zeichenroutine dahingehend verbessern, dass sie keine Zellen zeichnet, die sich nicht verändert haben. Wir könnten auch den Algorithmus beschleunigen, ohne Genauigkeit einzubüßen.

Sie können mit Queues sehr viel Geschwindigkeit gutmachen und das Ansprechverhalten verbessern, aber das hat seinen Preis. Sie müssen sicherstellen, dass es nicht versehentlich zu unerwünschtem Verhalten kommt, und Sie müssen sichergehen, dass die zusätzliche Komplexität der Queues auch wirklich zu messbaren Verbesserungen führt.

Lesen Sie Mike Ashs vierteilige Serie über Grand Central Dispatch und seine dazugehörigen Postings.[1] Drew McCormack bietet ebenfalls eine schöne Einführung in GCD an.[2]

1 *http://mikeash.com/pyblog/?tag=gcd*
2 *http://www.macresearch.org/cocoa-scientists-xxxi-all-aboard-grand-central*

Kapitel 27

Frisch ans Werk

Sie haben eine solide Einführung in die Umgebung der Mac- und iPhone-Entwicklung erhalten. Sie kennen das Programmieräquivalent zu einem guten Lokal und wissen, wie Sie mit dem Bus in die Stadt kommen. Wir haben nicht die Speisekarten und Preise aller Lokale aufgeführt, und auch nicht den ganzen Busfahrplan. Diese Details werden sich ändern, aber Sie können sie nachschlagen, wenn Sie sie brauchen.

Sie haben eine ganz schöne Reise hinter sich. Sie haben erfahren, wie man die grundlegenden Entwicklungswerkzeuge für Mac OS X- und iPhone-Apps nutzt. Sie haben die Besonderheiten von Cocoa und Objective-C kennengelernt. Am wichtigsten ist aber, dass Sie gelernt haben, wie Sie das meiste aus dieser Plattform herausholen, indem Sie die Techniken und Entwurfsmuster benutzen, die den Kern der Cocoa-Entwicklung bilden.

27.1 Was ist mit ...

Es gibt viele Themen, die ich eigentlich ins Buch mitaufnehmen wollte, aber im Endeffekt dann doch außen vor gelassen habe. Ich wollte mich in diesem Buch auf die Fertigkeiten konzentrieren, die Sie zum Programmieren benötigen, nicht auf die APIs. Es wird bestimmt Themen geben, die Sie hier gern gesehen hätten.

Ich hatte zum Beispiel ein einfaches Webservices-Beispiel entwickelt, in dem Sie einen einfachen Twitter-Client aufgebaut hätten. Als ich es mir noch einmal ansah, war es mir dann aber doch zu API-lastig. Sie hätten erfahren, wie man synchrone und asynchrone Requests sendet und wie man das in der Antwort enthaltene XML verarbeitet. Sie hätten sogar gelernt, wie man auf eine Authentifizerungsanfrage antwortet. Das

ähnelt einem Beispiel aus unserem iPhone SDK-Buch und dem, was wir in unserem iPhone Studio-Kurs vermitteln, aber es passt nicht in dieses Buch. Abgesehen von einigen neuen Klassen und Methoden waren Delegates die wesentliche Technik in diesem Beispiel.

In habe auch das Core Data-Beispiel auf das iPhone portiert. Das iPhone kennt keine Bindungen, weshalb Sie die Daten selbst abrufen und in einem Tabellen-View darstellen müssen. Obwohl die Einstellungen für Core Data in diesem Fall andere sind, werden im Wesentlichen solche Techniken zum Auffüllen von Tabellen verwendet, die Sie schon lange vor Core Data kennengelernt haben. Es gab also keinen guten Grund dafür, dieses Material zu behandeln.

Und so ging es weiter. Das Buch hätte ganz leicht doppelt so dick sein können, aber bei jedem Beispiel habe ich nach seiner Daseinsberechtigung gefragt. Bietet es wirklich etwas Neues? Häufig zuckten die fraglichen Kapitel mit den Schultern und sagten: „Ich glaube nicht."

Es gab eine Ausnahme. Ich habe monatelang mit mir gerungen, ob ich das Buch mit einem Kapitel über OpenCL abschließen sollte. Ich habe mir am Ende überlegt, dass es sich dabei um ein Nischenthema handelt. Tatsächlich hat mich diese Entscheidung dazu gebracht, das Kapitel zu Grand Central Dispatch so anzupassen, dass es die entsprechenden Grundlagen mitbehandelt.

Wenn Sie nun also sagen, „Sie hätten ein Kapitel zu … aufnehmen können", dann stimme ich Ihnen zu. Ich hätte es aufnehmen können. In vielen Fällen habe ich darüber nachgedacht, doch letztlich entschieden, dass das Buch durch das zusätzliche Material nicht besser geworden wäre. Ich wollte die Zahl der Abstecher minimieren, die wir auf unserer Reise durch die Cocoa-Entwicklung unternehmen mussten.

27.2 Wie geht es weiter?

Es ist an der Zeit, das Buch beiseite zu legen und mit der Entwicklung einer realen Anwendung zu beginnen.

Bevor Sie sich hinsetzen und eine Zeile Code schreiben, sollten Sie ganz genau festlegen, was Sie da schreiben und für wen es gedacht ist. Können Sie sich jemanden vorstellen, der Ihre Anwendung nutzt? Was macht er damit? Wann nutzt er sie? Welches Problem wird gelöst? Wie erklären Sie dem Anwender, warum er ihre Anwendung herunterladen und dafür bezahlen sollte?

Das klingt nach wenig, aber wenn Sie sich den vorigen Absatz ansehen, werden Sie erkennen, wie nützlich es ist. Ich wusste genau, für wen dieses Buch gedacht ist und was ich Ihnen mit ihm an die Hand geben will. Das machte mir die schweren Entscheidungen leichter. Identifizieren Sie Ihre Zielgruppe und was sie von Ihrer Anwendung hat.

Sie sollten jetzt immer noch keinen Code entwickeln. Skizzieren Sie zuerst die Bildschirmseiten und stellen Sie sich den Benutzer vor, der von Seite zu Seite navigiert und mit der Anwendung interagiert. Dieser Papier-Prototyp erscheint Ihnen vielleicht überflüssig, aber er hilft dabei, den Fluss Ihrer Anwendung zu durchdenken. Es ist wesentlich einfacher, Probleme zu beheben, bevor man mit der Entwicklung von Code und dem Aufbau von Nibs beginnt.

Während dieser ersten Schritte sollten Sie darauf achten, keine Wörter zu benutzen, die nur Programmierer verwenden. Reden Sie nicht über den Einsatz von Delegates, beschreiben Sie nicht das Modell und sprechen Sie nicht über separate Nibs. An diesem Punkt sollten alle Gedanken in den Begriffen abgefasst sein, die die Person verwendet, die später Ihre Anwendung benutzen soll. Das ist die am schwierigsten zu befolgende Regel; die Idee dahinter ist, dass man sich an diesem Punkt nicht um die Implementierung kümmert. Im Moment stellen wir nur sicher, dass wir etwas haben, das zu implementieren sich lohnt.

Danach sind Sie so weit, Xcode zu starten und mit der Programmierung zu beginnen. Wenn Sie einen Teil fertig haben, dann übergeben Sie ihn den Testern. Sie selbst sind zu nah an der Anwendung dran. Sie wissen ja genau, dass Sie in ein das und das Untermenü wechseln müssen, um etwas Bestimmtes zu finden – wenn die Tester das nicht hinbekommen, müssen Sie etwas daran ändern. In den meisten Fällen soll sich Ihre Anwendung so verhalten wie andere Mac- oder iPhone-Anwendungen auch.

Während der Entwicklung werden einige Ihrer Bemühungen in Sackgassen enden. Sie werden nicht in der Dokumentation finden, was Sie suchen. Es tritt ein Fall ein, der weder in diesem noch in den anderen Büchern behandelt wird, die Sie im Regal stehen haben. Bewaffnen Sie sich selbst mit Mike Ashs Posting „Getting Answers" und besuchen Sie Apple und unabhängige Foren und Mailinglisten.[1]

1 http://www.mikeash.com/getting_answers.html

Feilen und polieren Sie, aber denken Sie an Steve Jobs' berühmte Mahnung: „Real artists ship."[2]

Ich freue mich auf das, was Sie so auf die Beine stellen werden!

27.3 Danksagungen

Ich konnte mich nicht so recht entscheiden, wo in diesem Buch die Danksagungen hingehören. Ich weiß, dass sie üblicherweise am Anfang stehen, aber in diesem Buch geht es um Ihre Reise, nicht um meine. Ich habe weder den Anfang noch das Ende dieser Reise gesehen. Wenn das hier ein GCD-fähiges Programm wäre und kein Buch, dann hätte ich das Buch in der Haupt-Queue vorgehalten und die Danksagungen an einen nebenläufigen Thread mit hoher Priorität gesendet. Diese Danksagungen stehen also nicht etwa am Ende, weil Sie nicht wichtig wären.

Zuerst danke ich Kimberli Diemert und Margaret Steinberg. Niemand ist mir wichtiger als meine Frau Kim und meine Tochter Maggie. Ohne ihre Hilfe, Unterstützung und die tägliche Dosis Realität hätte ich nichts fertiggebracht.

Zweitens danke ich Dave Thomas dafür, dass er dieses Buch lektoriert hat. Das war ein Geschenk, von dem ich auf jeder Seite dieses Buches profitiert habe. Er hat Vorschläge zu Text und Code gemacht. Alles, was Sie an diesem Buch nicht mögen, ist definitiv seine Schuld.

Das war natürlich ein Scherz. Es ist wahrscheinlich Andy Hunts Fehler.

Ja, das war auch nur Spaß. Außer dafür, dass sie das Buch lektoriert haben, möchte ich Dave und Andy dafür danken, dass sie mir eine Heimat als Autor und Lektor gegeben haben und mir die Tools zur Verfügung gestellt haben, die das Schreiben und die Pflege eines technischen Buches ermöglichten.

Dank geht an Mike und Nicole Clark, die mir erlaubt haben, in den Pragmatic Studios Cocoa- und iPhone-Programmierung zu unterrichten. Dank an die vielen Studenten der Studios, die mich mit vielen Rückmeldungen zum Material versorgt haben. Dank an die Freunde und Kollegen Craig Castelaz, Chris Adamson, Bill Dudney, Eric Freeman, Scott Kovatch und Dee Wu für ihre hilfreichen Kommentare während dieses Projekts.

2 Frei übersetzt etwa „Wahre Künstler liefern auch aus"; siehe *http://www.folklore.org/StoryView.py?story=Pirate_Flag.txt*

Dank an die technischen Korrektoren und die Leser, die Fehler gemeldet haben, während dieses Buch im Betastadium war. Dank an die Leute bei Apple, deren Namen ich nicht nennen darf, da sie sonst in Schwierigkeiten geraten würden. Zwei von ihnen sind besonders zu erwähnen: Sie luden mich vor Jahren zum Essen ein, um dafür zu sorgen, dass ich auch wirklich begriff, dass ich von Java zu Objective-C wechseln musste, wenn ich auch weiterhin Code für diese Plattform entwickeln wollte. Zum Schluss mein Dank an die Leute bei Apple, die diese wundervolle Plattform entwickelt haben. Programmieren mit Cocoa und Objective-C ist ein wahres Vergnügen.

27.4 Widmung

Dieses Buch ist meinem Freund James Duncan Davidson gewidmet.

Ich traf Duncan auf einer Apple Worldwide Developer Conference (WWDC), als sie noch in San Jose stattfanden. Er recherchierte für sein Buch *Learning Cocoa* [DA02]. Wenn es nach mir gegangen wäre, hätte er dieses Buch geschrieben und ich hätte es lektoriert. Ich habe versucht, mich an den Geist seines Buches zu halten.

Duncan traf Kim (meine Frau) auf einer Mac-Geek-Reise und wurde schnell zu einem Freund der Familie. Meine Töchter lernte er kennen, direkt bevor wir beide zur Mac Hack fuhren. Mit anderen Worten hatten die ersten Begegnungen meiner Familie mit diesem Mann immer etwas mit Mac-Events zu tun.

Meine Tochter Maggie raunte ihren Freunden mit gedämpfter Stimme zu: „Der ist ganz berühmt. Er hat Ant und Tomcat erfunden". Sie vermurkste auch immer absichtlich seinen Namen und nannte ihn James David Duncanson.

Ich fand sein Verhältnis zu meiner jüngsten Tochter toll. Egal ob in unserem Haus oder wenn wir ihn in Portland besuchten oder ihn in San Francisco trafen, es war immer ein ganz besonderes Geben und Nehmen. Sie neckte ihn und flirtete mit ihm, und er sprach mit ihr in einer Weise, die sowohl ihrem Alter angemessen als auch respektvoll war. Ja, war. Als Elena starb, kam Duncan und kümmerte sich um uns. Er ließ alles stehen und liegen und blieb bei uns, als die Beerdigung vorbei war, und er sorgte dafür, dass wir alles Notwendige hatten.

Vor ein paar Jahren sprach ich Duncan darauf an, ob er eine aktualisierte Fassung seines Cocoa-Buches schreiben wolle. Er lehnte ab und schlug vor, dass ich es schreiben solle. Ich war nervös. Ich musste daran denken, dass er der Cleveland Java Users Group einmal erzählt hatte, dass er sich für einen OO-„Rocker" gehalten hatte, während er bei Sun an Java arbeitete; bei der Arbeit mit Objective-C und Cocoa hatte er aber gemerkt, dass er das damals gar nicht gewesen war. Das war für mich der Leitgedanke beim Schreiben dieses Buches: Ich wollte nicht einfach die Cocoa-APIs durchgehen und Ihnen nur die Werkzeuge erklären, sondern Ihnen die Techniken zeigen, mit denen Sie zu einem OO-Rocker werden würden.

Danke, James!

Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z