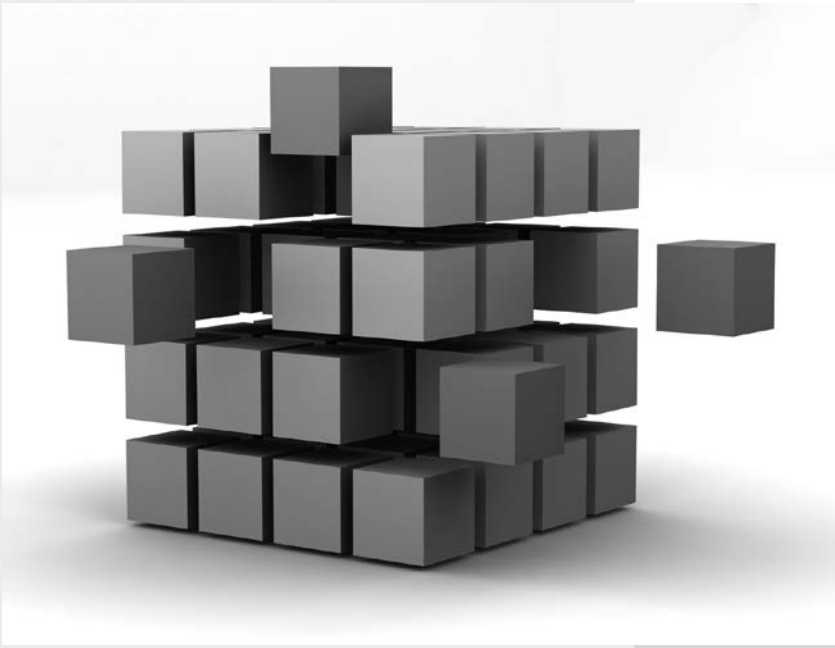
Kapitel

5

Die Viewschicht



Sie haben jetzt einen Einblick in alle wichtigen allgemeinen Elemente von Cocoa gewonnen. Hiermit beginnen wir eine Reise in speziellere Themengebiete, zunächst in die Viewschicht.

In diesem Kapitel geht es mir darum, dass Sie die verschiedenen Elemente der Viewschicht kennenlernen und ihre Beziehung zueinander verstehen. Sie sollen diese also sinnvoll anwenden können, insbesondere was die Einstellungen angeht. Außerdem möchte ich Ihnen einen Überblick über dieses Thema als Eintrittskarte in fortführende Dokumentation geben.

HILFE

Wir arbeiten am Projekt aus Kapitel 2 weiter. Laden Sie das Converter-11 von der Webseite herunter.

Damit wir nicht unser mühsam zusammengebautes Fenster zerstören, müssen Sie erst einmal ein weiteres Fenster in den Document.xib ziehen. Öffnen Sie daher diese Datei. Das finden Sie in der Object-Library unter dem Namen *Window* (notfalls suchen). In dem Identity-Inspector geben Sie bitte als Namen *Experimentierfenster* ein, damit es nicht zu Verwechselungen kommt. Auch hier verweise ich für Fälle der Verzweiflung auf die How-Tos auf der Webseite zum Buch – oder Sie lesen noch einmal den entsprechenden Abschnitt in Kapitel 2.

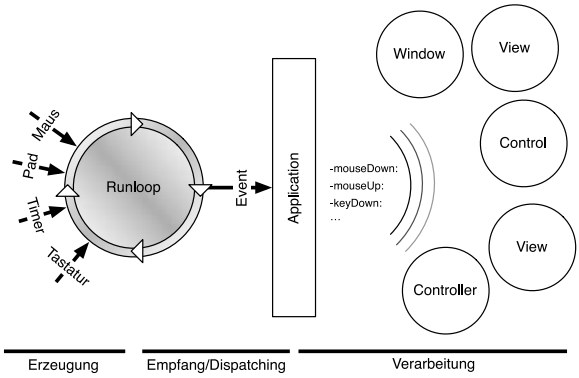
Achten Sie bitte noch im Attributes-Pane des Inspectors darauf, dass der Haken bei *Visible at Launch* gesetzt ist. Dies sorgt dafür, dass beim Laden des Nibs das Fenster auch angezeigt wird.

5.1 Grundlagen

Schauen wir uns zunächst das Gesamtsystem an. Die wesentlichen Elemente, die in der Viewschicht eine Rolle spielen, sind Fenster von der Klasse `NSWindow` und Views von der Klasse `NSView`.

5.1.1 Responder als Basisklasse

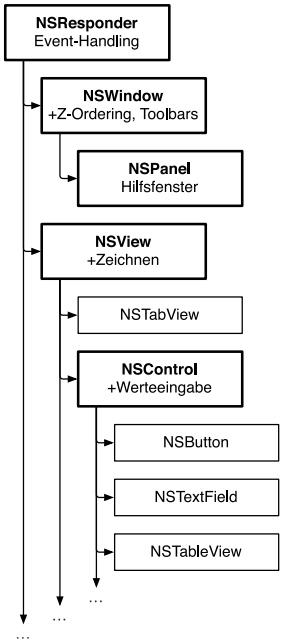
Die beiden Klassen `NSWindow` und `NSView` stammen nicht unmittelbar von `NSObject` ab, sondern erben von der Klasse `NSResponder`. `NSResponder` ist dafür verantwortlich, Nachrichten des Benutzers (letztlich also des Systems) auszuführen. Sie hatten ja bereits gesehen, dass etwa Buttons eine Nachricht klick erhalten. Nein, in Wirklichkeit existiert eine Methode mit diesem Namen nicht, sondern Methoden wie `–mouseDown:`, `–mouseUp:` usw. Diese Methodenaufrufe werden durch den Responder verarbeitet. Man nennt diese Systemnachrichten »Events«, den Gang habe ich kurz skizziert:



Die grobe Struktur des Event-Versandes. Am Ende steht ein Responder.

GRUNDLAGEN

Die Verteilung von derlei Nachrichten des Systems in der Applikation ist nicht einfach zu verstehen und erfordert bereits gute Kenntnisse von Cocoa, weil verschiedene Klassen beteiligt sind und zudem die Verteilung je nach Art des Events auf unterschiedliche Weise erfolgt. Glücklicherweise verhält es sich jedoch umgekehrt so, dass man sich allermeist nicht darum kümmern muss, weil bereits alles funktioniert, wie man es benötigt. Ich habe daher die Einzelheiten in den zweiten Band gestopft. Ein bisschen werden wir aber auch schon hier damit experimentieren.



Es gibt übrigens weitere Responder. Denn auch Objekte, die nicht auf dem Bildschirm sichtbar sind, also keine Windows oder Views sind, können Nachrichten vom System empfangen. So kann es sinnvoll sein, eine Operation wie Undo nicht von einem View ausführen zu lassen, sondern auf einer höheren Ebene: Wenn Sie sich gerade bei der Eingabe in einem Textfeld befinden, erwarten Sie etwa bei einem Klick auf *Undo*, dass Ihre letzte Operation in diesem Feld zurückgenommen wird. Dies bezieht sich also nur auf das Textfeld als View. Haben Sie gerade in eine Tabelle einen neuen Eintrag eingefügt, würden Sie eher erwarten, dass dieser wieder entfernt wird. Dies betrifft die Controllerschicht. Und wenn im Menü einer Applikation *Quit* (*Beenden*) ausgewählt wird, bezieht sich das offenkundig auf die Applikation als Ganzes. Sie sehen also, dass die gleiche Nachricht von Instanzen unterschiedlicher Klassen gefangen werden kann. Diese Klassen sind fast immer Subklassen von `NSResponder`, wie auch unsere Klassen, aber eben nicht nur.

Ein Überblick der Viewklassen als Subklassen von NSResponder

Fenster

Ausgangspunkt für jede Bildschirmausgabe sind die Fenster. Fenster reservieren einen Bereich des Bildschirms für die Ein- und Ausgabe. Außerdem sind Fenster hintereinander angeordnet, so dass bestimmt werden kann, wie sie sich gegenseitig (teilweise) verdecken und welche ihrer Flächen sichtbar sind. Man nennt dies das Z-Ordering. Außerdem können sie Toolbars (Symbolleisten) verwalten und sind dafür verantwortlich, Events unter ihren Views zu verteilen.

Es existiert noch eine wichtige Subklasse von `NSWindow`, nämlich `NSPanel`, welches wiederum Subklassen hat. Ein Panel ist ein Hilfsfenster wie ein Inspector, das sich üblicherweise durch eine schmalere Titelleiste auszeichnet und zudem verschwindet, wenn die Anwendung in den Hintergrund gerät.

Keine Fenster stellen indessen sogenannte Drawer (`NSDrawer`) dar, die ebenfalls von `NSResponder` abgeleitet sind. Bei diesen handelt es sich um die etwas aus der Mode gekommenen Schubladen, die manche Fenster haben. Sie sind jedoch stets mit einem Fenster verbunden.

Views

Instanzen der Klasse `NSView` nehmen die tatsächliche Kommunikation (Ein-/Ausgabe) mit dem Nutzer vor. `NSView` addiert daher zur Klasse `NSResponder` vor allem Methoden, die mit dem Zeichnen und dem Emfpang von Nutzerereignissen zusammenhängen. Außerdem verwaltet ein View sogenannte Subviews, ordnet also Views hierarchisch an. Views besitzen ebenfalls einen bestimmten Bildschirmbereich, der sich immer innerhalb eines Fensters befindet. Keine Sorge, wir gehen das gleich im Einzelnen durch.

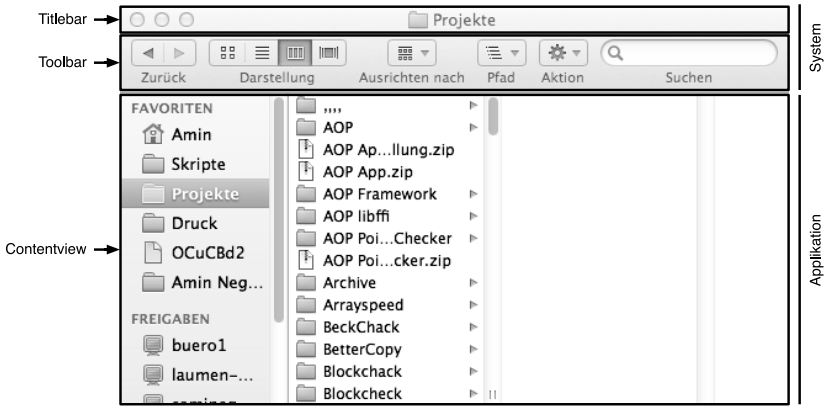
POWER

Hier sieht man übrigens einen deutlichen Unterschied zum Framework Cocoa Touch auf dem iPhone: Dort gibt es keine überlappenden Fenster wie aus einem Desktop-Computer, weshalb das Fenster eine Subklasse von `NSView` ist. Wie Sie gleich sehen werden, verhält sich dies unter Cocoa völlig anders.

Die Aufgabenteilung zwischen Fenstern und Views

Zwei Bereiche des Fensters müssen auseinandergehalten werden: Der Bereich, der etwa die Titelleiste oder eine Toolbar enthält, gehört dem System. Der untere Bereich des Fensters gehört dagegen der Anwendung, also Ihnen als deren Programmierer.

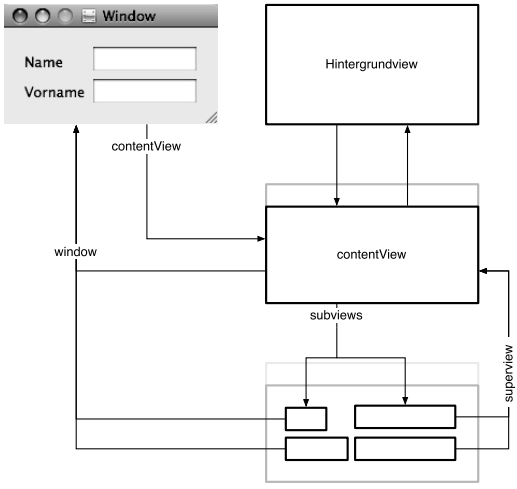
Fenster, und das ist wichtig zu verstehen, zeichnen jedoch nicht selbst. Sie bedienen sich der Views, um Ausgaben zu erledigen. Dabei existiert für jedes Fenster zunächst ein geheimes Hintergrundview, welches sozusagen das Fenster abdeckt. Dies ist deshalb geheim, weil es keine offizielle Methode gibt, dieses View zu ermitteln.



Teile und herrsche: die beiden Bereiche und ihre Funktion am Beispiel des Finders

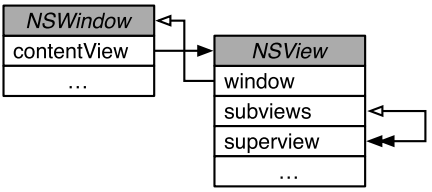
Die Unterteilung der Bereiche wird durch sogenannte Subviews bewerkstelligt. Jedes View kann nämlich wiederum Unterviews haben. So befinden sich in dem Hintergrundview wiederum Views für Elemente des Systems (Titelleiste, Toolbar, Fensterknöpfe usw.) und – dieses View ist für uns wichtig – wiederum ein sogenanntes Contentview, das den Applikationsbereich markiert. Es ist für jedes Fenster mittels der Methode `–contentView` abrufbar. Dieses Contentview stellt sozusagen das oberste View in der Hierarchie dar, so weit es unsere Sicht der Dinge als Anwendungsprogrammierer angeht.

Die Views, die Sie als Elemente in das Fenster ziehen, sind dann wiederum Subviews des Contentviews. Ein Beispielfenster, bei dem ich allerdings nicht alle Beziehungen eingezeichnet habe:



Unsere Elemente sind wiederum Subviews des Contentviews.

Hierbei ist zu beachten, dass ein View seine Subviews im Sinne der Speicherverwaltung hält (`strong`), der Superview-Zeiger dagegen zur Vermeidung eines Retain-Cycles einen schwachen Verweis darstellt.



Formelle Beziehung zwischen Window und Contentview sowie der Views untereinander

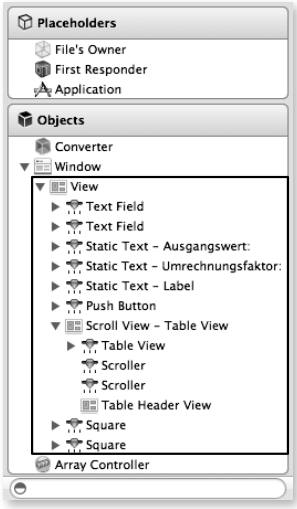
Diese Struktur ab dem Contentview ist wie erwähnt für den Anwendungsprogrammierer dokumentiert und offen. Die wichtigsten Methoden zur Verwaltung dieser Struktur:

- `–subviews` liefert uns ein Array der Subviews eines Views. In obiger Zeichnung würden wir also die vier Textfelder erhalten, wenn wir diese Nachricht an das Contentview versenden.
- `–superview` gibt das jeweilige Superview zurück. Wenn wir also oben in dem Beispiel diese Methode auf ein Textfeld anwenden, so erhalten wir das Contentview.
- `–window` liefert für jedes View (nicht nur das Contentview) das Fenster.
- `–addSubview` fügt das als Parameter übergebene View dem Empfänger der Nachricht hinzu.
- `–removeFromSuperview` entfernt den Empfänger aus der Kette der Subviews.

AUFGEPASST

Wenn Sie manuelle Speicherverwaltung verwenden, existiert hier ein beliebter Fehler: Ein Subview wird regelmäßig nur von dessen Superview gehalten. Sobald Sie ihn also entfernen, wird er in der Regel gelöscht und ist nicht mehr ansprechbar. Um das zu ändern, müssen Sie bei MRC selbst eine Retain-Nachricht vor dem Entfernen aus der Viewhierarchie senden und eine Release-Nachrichten, wenn Sie das View nicht mehr benötigen. Bei automatischem Reference-Counting haben Sie in der Regel bereits einen Zeiger auf das Subview, das ja standardmäßig `strong` ist. Dieser hält dann das View, so dass man sich keine weiteren Gedanken machen muss.

Wir können uns im Interface Builder die Hierarchie der Views teilweise anschauen:



Die Hierarchie der Views im Interface Builder

Öffnen Sie hierzu das Hauptfenster *Document.xib.* Öffnen Sie dann die Disclosures wie in der Abbildung angezeigt. Sie können jetzt die Elemente sehen, die in dem Contentview liegen. Achten Sie darauf, dass das Tableview als *Scroll View – Table View* angezeigt wird. Welche Bewandtnis es damit hat, sehen Sie, wenn wir gleich die entsprechenden Klassen besprechen. Manchmal ist die Welt nämlich komplizierter, als sie uns der Interface Builder zeigt. Das können Sie bereits daran erkennen, dass das Hintergrundview gar nicht erst in der Aufstellung auftaucht. (Er ist ja auch geheim.)

Programmieren wir mal etwas herum. Erweitern wir unsere Controller-Klasse Converter um eine weitere Action. Zunächst Converter.h:

```
@interface Converter : NSObject
@property (weak) IBOutlet NSTextField *inputTextField;
@property (weak) IBOutlet NSTextField *factorTextField;
@property (weak) IBOutlet NSTextField *outputTextField;

- (IBAction)calculate:(id)sender;
- (IBAction)playWithViews:(id)sender;
@end
```

Diese Action-Methode muss dann freilich noch in der Implementierungsdatei Converter.m nach `-calculate:` eingefügt werden. Beginnen wir mit etwas Leichtem:

```
- (IBAction)playWithViews:(id)sender
{
   NSLog( @"Titel: %@", [sender title] );
}
@end
```

Fügen Sie nun eben im Interface Builder in das Document-Fenster – also dem bereits vor diesem Kapitel existierenden Fenster – einen weiteren Button ein, den Sie mit *Spielkind* beschriften. Durch [ctrl]-Ziehen verbinden Sie diesen mit der neuen Action-Methode unseres Converters. Kompilieren und starten. Nach einem Klick auf den neuen Button erscheint im Log-Fenster folgende Ausgabe:

```
>… Titel: Spielkind
```

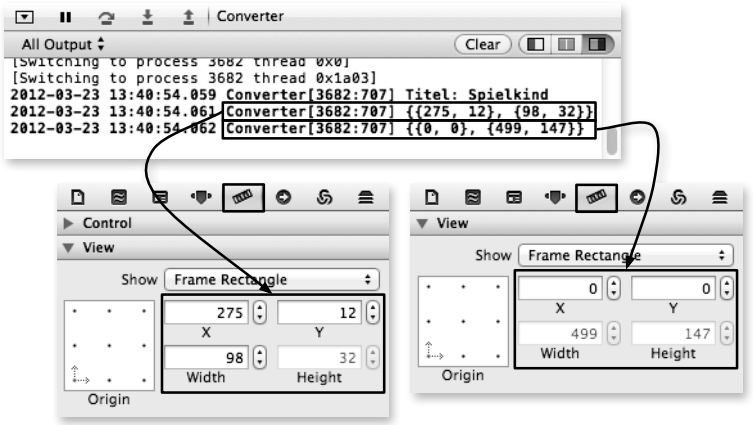
Nun, die Sache ist einfach: Eine Action-Methode bekommt als Parameter mitgeliefert, wer die Action ausgelöst hat. Das ist in unserem Falle der Button. Auf diesen Button können wir die Methode `–title` anwenden, um die Beschriftung zu erhalten. That's it!

Wenn aber `sender` der Button ist, so können wir auch das Superview abfragen. Dieses müsste dann das Contentview sein. Probieren geht über Studieren:

```
– (IBAction)playWithViews:(id)sender
{
   NSRect frame;
   NSLog( @"Titel: %@", [sender title] );

   frame = [sender frame];
   NSLog( @"%@", NSStringFromRect( frame ) );

   NSView* contentView = [sender superview];
   frame = [contentView frame];
   NSLog( @"%@", NSStringFromRect( frame ) );
}
@end
```



Die Werte im Log und im Interface Builder stimmen überein.

Nach dem Start und einem Klick auf den Button erhalten Sie im Log verschiedene Werte. Dies sind die Position und die Größe des Buttons und des ihn umgebenden Views. Diese Koordinaten beziehen sich auf das jeweilige Superview: Also, die Koordinaten des Buttons beziehen sich auf das Contentview, die Koordinaten des Contentviews beziehen sich auf das Hintergrundview. Sie können sich auch dies im Size-Inspector des Interface Builders anzeigen lassen und die Werte aus Ihrem Log vergleichen. Bei Ihnen weichen diese Werte selbstverständlich von den hier abgebildeten ab, sollten aber in sich übereinstimmen.

HILFE

Sie können das Projekt in diesem Zustand als Converter-12 von der Webseite herunterladen.

Sie können übrigens in der großen, linken Fläche die Ausrichtung des Koordinatensystems ändern. Dies gilt aber nur für die Anzeige der Werte im Interface Builder. Probieren Sie es aus und starten Sie das Programm erneut. Im Log haben sich die Koordinatenwerte nicht verändert.

POWER

Jedes View hat neben diesen Koordinaten, die seine Lage und Größe im Superview angeben, ein weiteres Rechteck (Bounds, Boundaries), welches das Koordinatensystem im Inneren beschreibt. Wir benötigen dies in diesem Band nicht, besprechen es aber im zweiten Band, wenn Sie eigene Views programmieren werden.

Damit Sie mir wirklich glauben, dass die Views im Interface Builder Subviews des Contentviews sind, verändern wir die Methode:

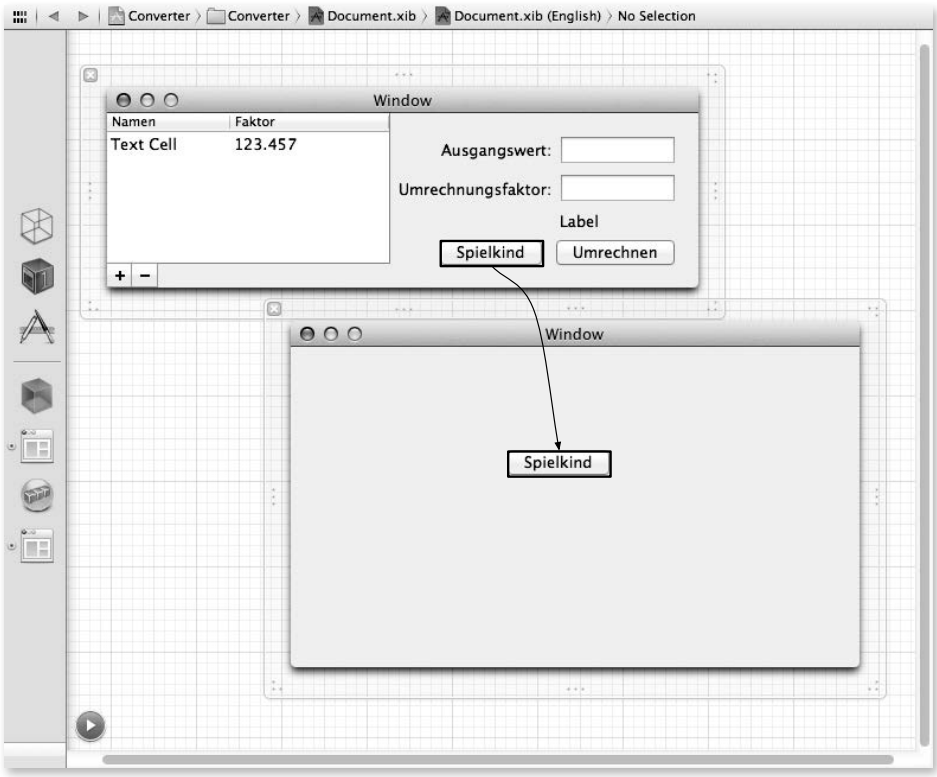
```
- (IBAction)playWithViews:(id)sender
{
   NSRect frame;

   NSView* contentView = [sender superview];
   for(NSView* subview in [contentView subviews]) {
      frame = [subview frame];
      NSLog(@"%@:", NSStringFromClass([subview class]));
      NSLog(@"%@", NSStringFromRect(frame ));
   }
}
@end
```

Hier wird also zunächst das Contentview geholt und dann mit einer For-in-Schleife durch alle Subviews dieser Ebene iteriert. Für jedes Subview werden der Name der Klasse und die Koordinatenangaben gedruckt. Vergleichen Sie die Ausgabe im Log bitte wieder mit den Werten im Interface Builder.

HILFE

Sie können das Projekt in diesem Zustand als Converter-13 von der Webseite herunterladen.



Werden zwei Fenster in der Objektleiste nacheinander angeklickt, erscheinen sie beide.

Zuletzt ziehen Sie bitte den Spielkind-Button in das neue Experimentier-Fenster. Dazu klicken Sie einfach in der Objektliste auf das Experimentierfenster, um es zusätzlich einzublenden. Ist Ihnen das zu unübersichtlich, bleibt die Möglichkeit, mit Copy-and-paste zu arbeiten. Sie können überprüfen, dass die Verbindung zur Action-Methode hierbei erhalten bleibt. Die Action-Methode `–playWithView:` leeren Sie bitte, ohne sie zu löschen.

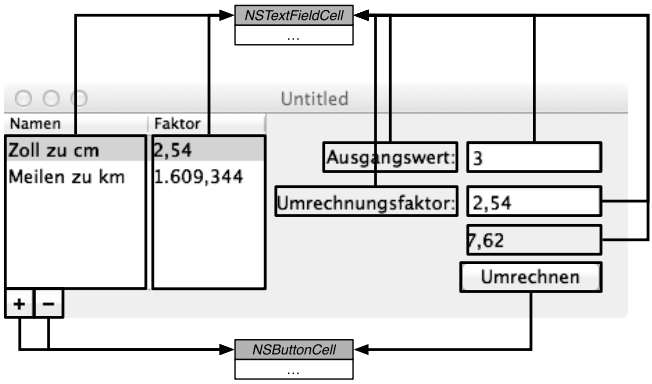
5.1.2 Views und Cells

»Die Geschichte der weiblichen Menstruation ist eine Geschichte voller Missverständnisse«, lautete mal ein Werbeslogan. Ich kann das nicht wirklich beurteilen, weiß aber, dass die Geschichte von Views und Cells ganz sicherlich voller Missverständnisse steckt.

Wie Sie bereits gesehen haben, sind Views Bereiche, die eine Lage haben und in die gezeichnet werden kann. Außerdem können sie aufgrund von Events tätig werden.

Dann stellt sich freilich die Frage, warum es überhaupt Cells gibt: In unserem Dokumentfenster wird in den Textfeldern Text dargestellt. Deshalb muss irgendwo in der Klasse `NSTextField` Code existieren, der Text zeichnet, Mausklicks entgegennimmt usw. Gut, das war jetzt noch keine intellektuelle Herausforderung. Aber jetzt schauen wir etwa auf ein Tableview: Auch dieser kann Text darstellen. Also muss jetzt auch ein Tableview Code haben, der Text darstellt, zum Editieren Tastendrücke annimmt usw. Das geht noch weiter: Ein Imageview stellt ein Bild dar. Auch in einem Tableview kann aber ein Bild enthalten sein. Xcode zeigt in dem Projektnavigator auch Bildchen an.

Damit man jetzt nicht den bereits in Textfields enthaltenen Code für Tableviews erneut programmieren muss, liegt es nahe, das Zeichnen (und die Entgegennahme von Events) vom View selbst zu isolieren und in eine eigene Klasse Cells zu packen. Die einzelnen Views bedienen sich dann der Cells. Viele Views wie ein Button oder ein Textfeld haben dann ihre Cell, während andere Views wie ein Tableview verschiedene Cells kombinieren. Dies spart Code und macht die Sache übersichtlicher.



In verschiedenen Views, aber doch immer dasselbe: Cells modularisieren.

Meist werden Cells von Controls benutzt. Das System der Cells lässt sich jedoch auch allgemein bei Views anwenden. Dabei gilt regelmäßig, dass ein Control genau eine Cell hat. Zwingend ist das nicht.

Kommen wir damit zum zweiten wichtigen Punkt und damit zum eigentlichen Missverständnis: In einem Tableview verfällt man leicht auf den Gedanken, dass die einzelnen Felder in einer Spalte jeweils eine Cell darstellen, also etwa eine Textfieldcell für das Feld *Zoll zu cm*. Dies ist grundfalsch.

Cells haben nämlich keinen Zeichenbereich, für den sie zuständig sind. Dann wären es ja Views. Anders: Ein solches Tableview wie in der Abbildung hat in der Regel eine einzige

Textfieldcell für sämtliche Zeilen in einer Spalte. Wenn es die Mitteilung bekommt, sich neu zu zeichnen, dann wird damit die entsprechende Textfieldcell beauftragt und ihr mitgeteilt, wo sie zeichnen soll. Der Zeichenbereich ist also nicht eine Eigenschaft der Cell, sondern ein Parameter für die konkrete Malerei. Die richtige Parallelvorstellung von Cells in der Wirklichkeit ist also ein Stempel oder eine Zeichenschablone, die an die richtige Stelle geschoben werden. Der Code in `NSTableView` sieht dann symbolisch so aus:

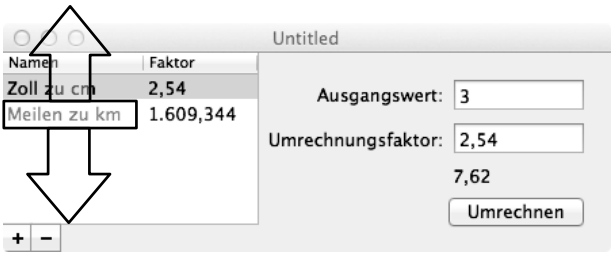
```
// zeichne eine Textspalte
NSTextFieldCell *cell = self.textFieldCell;
NSRect frame = …
for(rowIndex = startIndex; rowIndex <= endIndex; rowIndex++) {
    // Wert setzen
   cell.stringValue = …

   // Rahmen berechnen
   frame.origin.y = rowIndex * self.rowHeight;

   // Zeichnen
   [cell drawWithFrame:frame inView:self]
}
```

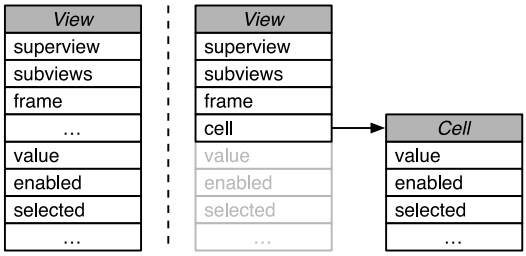
Sie sehen also, dass lediglich vor der Schleife einmal eine Cell geholt wird. Es wird dann keine Eigenschaft `frame` der Zelle gesetzt, sondern vielmehr das Rechteck als Parameter übergeben.

Ebenso sind Cells nicht Bestandteil der Viewhierarchie. Sie haben ihr View nicht einmal als Eigenschaft. Auch dies ist nur ein Parameter bei einer konkreten Aktion. Dasselbe gilt übrigens für die Verarbeitung von Events: Zwar sind Cells keine Responder, so dass sie selbst keine Events empfangen können. Das ist auch für Mausklicks einsichtig: Cells haben keinen festen Bereich auf dem Bildschirm, können also gar nicht wissen, ob gerade sie angeklickt wurden. Vielmehr ist es auch hier so, dass sich das View der Cell nur zur eigentlichen Verarbeitung bedient, nachdem das View die Nutzeraktion empfangen hat.



Wenn in der fünften Zeile ein Text gemalt werden muss, wird eine Textfieldcell an eben diese Stelle verschoben.

Viele Eigenschaften eines Views sind übrigens gar nicht dessen Eigenschaften, sondern die seiner Cell. Wir hatten etwa gesehen, dass man die Textfarbe einer NSTextField-Instanz setzen kann. In Wirklichkeit wird aber der Methodenaufruf nur an die mit der `NSTextField`-Instanz verbundenen Instanz der Klasse `NSTextFieldCell` weitergeleitet.



Nach außen verborgen: Cells implementieren einen Teil des Views.

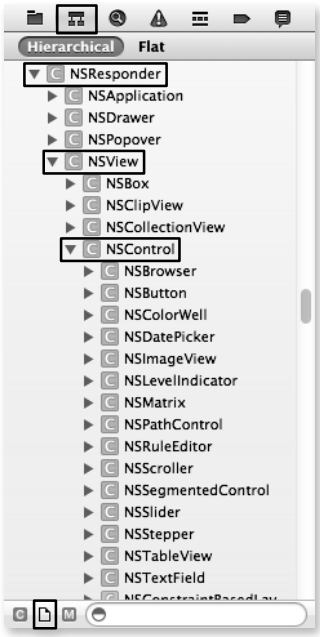
POWER

Ich erzähle das alles hier, weil Sie bei manchen Views, wie dem Tableview, die Cells einstellen können. Natürlich wollte ich Ihnen auch einen Einblick in die Struktur geben. In Band 2 werden wir eigene Cells programmieren und benutzen. Code, der sich außerhalb eines Views befindet, kommuniziert in aller Regel nur mit dem View.

Eine bedeutende Subklasse von `NSView` ist `NSControl`. Controls – die nichts mit Controllern zu tun haben – sind diejenigen Views, welche vor allem Benutzereingaben, insbesondere Texteingaben, zu einem Wert verarbeiten. Ob allerdings eine Viewklasse unmittelbar von `NSView` oder über den Zwischenschritt `NSControl` abgeleitet wurde, hängt bei Apple offenkundig vor allem von Praktikabilitätserwägungen ab. Eine scharfe Trennlinie lässt sich nicht erkennen.

Manchmal kommt offenbar auch Apple etwas durcheinander: Die Klasse `NSTabView` (besprechen wir später) wird etwa in dem Dokument über Controls angesprochen. Sie ist aber unmittelbar von `NSView` abgeleitet. Auch die Fortschrittsanzeige (`NSProgressIndicator`) ist so ein Fall.

Sie können sich übrigens die Klassenhierarchie anzeigen lassen, indem Sie in Xcode zum Symbol-Navigator wechseln und dort unten die Option *Show only project-defined symbols* ausschalten. Öffnen Sie den obersten Eintrag *NSObject* und scrollen Sie in der Auswahlliste links bitte bis zu *NSResponder*. Ebenfalls öffnen. Sie sehen jetzt die Subklassen von `NSResponder`, also all jene Klassen, die Ereignisse vom Benutzer empfangen können. Öffnen Sie nun den Eintrag *NSView*, um zu den Klassen dieses Kapitels zu gelangen. Hierin können Sie dann noch *NSControl* erweitern.



Die Klassenhierarchie der Responder im Class-Browser von Xcode

GRUNDLAGEN

Viele Eigenschaften von Objective-C führen dazu, dass man derlei verschachtelte Klassenhierarchien häufig nicht braucht. In aller Regel sind auch die Hierarchien in Cocoa sehr flach. `NSResponder` bildet da die Ausnahme. Im Kapitel über das Vorgehen bei der Programmierung einer eigenen Applikation gehe ich auf die verschiedenen Möglichkeiten ein, eine vorgefertigte Klasse zu erweitern. Selten muss man dazu ableiten.

5.1.3 Nib-Files

Nib-Files (Nibs) sind Ansammlungen von Objekten, die beim Laden des Nibs automatisch im Speicher angelegt werden. Sie müssen nicht nur aus der Viewebene stammen, sondern enthalten zuweilen auch Instanzen, deren Klasse der Controllerebene entspringen.

Es existieren mehrere Möglichkeiten, Nibs zu laden. Wir werden uns hier nur mit dem sogenannten Automatic-Support beschäftigen, also dem automatischen Laden. Darüber hinaus existieren noch andere Vorgehensweisen, die allerdings wesentlich unbedeutender sind und die zudem ein gewisses – sagen wir – veraltetes Verständnis von der manuellen Speicherverwaltung mit sich bringen.