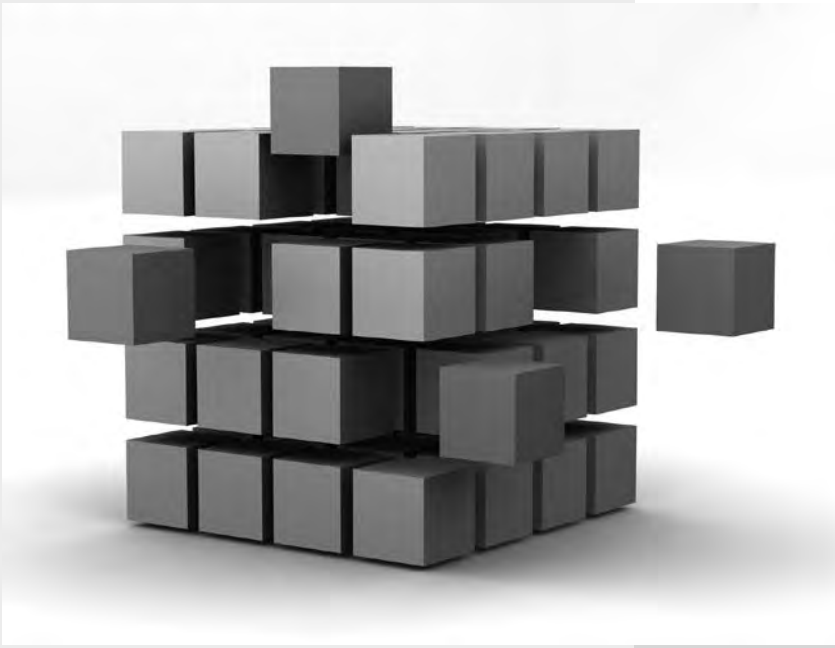
Kapitel

6

Die Controllerschicht



Steigen wir die Wendeltreppe des Programmiergrauens eine Stufe tiefer und begeben uns auf die Niederungen der Controllerschicht. Da Sie dann beide Partner der Beziehung »View zu Controller« kennenlernen, können auch einige offene Punkte aus dem letzten Kapitel geklärt werden. Auf der Controllerebene ist mehr Code angesagt, da eine Anwendung von ihren Controllern individuell geprägt wird. Aber mit etwas Übersicht bekommt man das in den Griff.

Da Controller eine sehr individuelle Angelegenheit sind, lassen sie sich im Vergleich zu Views schlechter kategorisieren. Dennoch gibt es typische Aufgabengebiete und vorgefertigte Controllerklassen. Es lässt sich also durchaus eine sinnvolle, wenn auch nicht immer scharfe Unterscheidung formulieren:

Zum einen haben wir Controller, die den Datenfluss besorgen. Hier sind zwei Gruppen zu nennen, die Sie zumindest schon dem Namen nach kennen:

- Bindings-Controller wie `NSArrayController`
- Data-Sources als eigene Klasse

Daneben gibt es Controller, die den Programmablauf (Kontrollfluss) implementieren:

- eigene Controller wie `Converter`
- Windowcontroller und Viewcontroller mit lokaler Funktionalität
- der Applikationscontroller als Delegate von `NSApplication`
- Notification-Observer als Empfänger asynchroner Überwacher

Zu der letzten Gruppe kann man auch `NSDocument` zählen. Deren Instanzen sind allerdings auch gleichzeitig Ausgangspunkt für unser Model. Umgekehrt werden Sie sehen, dass man die dort anzusiedelnde Funktionalität auch gut in Windowcontroller auslagern kann. Letztlich handelt es sich vielleicht um eine Zwischenschicht, die janusköpfig in beide Richtungen schaut. Ich bespreche die Klasse jedenfalls im Abschnitt über Models.

Aber im letzten Absatz befand sich noch eine wichtige Information: Im Kapitel 2 habe ich aus Vereinfachungsgründen zusätzliche Funktionalität stets in der Converter-Klasse implementiert. Irgendwie müssen wir das mehr strukturieren. Und auch darauf gehe ich ein.

6.1 Bindings-Controller, KVC, KVV und KVO

Das klingt ja schon in der Überschrift fürchterlich techi. Und das, wo Sie gerade aus dem gestalterischen View-Kapitel hierher gelangt sind. Ein Wechselbad der Gefühle!

6.1.1 Grundlagen

Nein, so schlimm ist es gar nicht: Die vier Konzepte bezeichnet man als Key-Value-Technologien.

Key und Key-Path

Wie Sie bereits gelernt haben, besitzen Entitäten Eigenschaften, die einen Namen haben. Im Rahmen von Key-Value-Technologien werden die Namen Key (Schlüssel) genannt. Sie stellen sich dann ein bisschen wie Dictionarys dar.

Bezeichnet ein Schlüssel eine Eigenschaft, die wiederum auf eine Entität verweist, ist die Eigenschaft also eine Relationship, so kann man das freilich fortführen und auf die bezogene Entität wiederum einen Schlüssel anwenden usw. Dies nennt sich Key-Path (Schlüsselpfad). Hier im Überblick beachten wir aber aus Vereinfachungsgründen diesen Fall nicht.

Zudem werden auch Operatoren unterstützt, die mit @ beginnen. Diese erlauben eine Zusammenfassung bei einer To-many-Relationship. Die wichtigsten Operatoren sind:

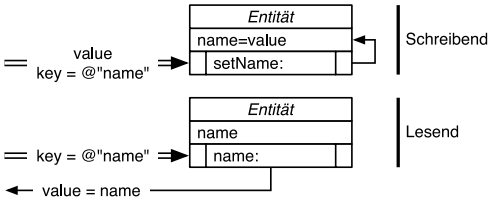
- `@count` `liefert` die Anzahl der bezogenen Objekte. Haben wir also eine Gruppe, die 5 Mitglieder über eine To-many-Relationship referenziert, so erhalten wir als Ergebnis des Schlüsselpfades `members.@count` den Wert 5.
- `@avg`, `@max`, `@min` errechnen aus einer Eigenschaft der bezogenen Objekte den jeweiligen Durchschnitt, Maximal- bzw. Minimalwert. Daher muss diese Eigenschaft hinter dem Operator angegeben werden. Besitzt also die Entität der Mitglieder eine Eigenschaft salary, die das Gehalt angibt, so würden man mit `members.@avg.salary` das Durchschnittseinkommen der Mitglieder erhalten. `@sum` errechnet die Summe.

Wir werden das später auch anwenden.

TIPP

Für komplexere Operationen existieren weitere Operatoren. So lässt sich etwa bei einer Liste von Personen aus der `To–many–`Relationship eine Liste von Vornamen erzeugen (`@unionOfObjects.firstName`) oder eine Liste von unterschiedlichen Vornamen (`@distinctUnionOfObjects.firstName`) – doppelte werden also nur einmal aufgenommen. Schließlich geht das Ganze mit einer weiteren To-many-Relationship noch weiter: Aus einer Liste von Gruppen, die wiederum jeweils auf eine Liste von Mitgliedern verweist, lässt sich eine Vereinigungsmenge der Mitglieder bilden, wobei auch wiederum hier entschieden werden kann, ob doppelte Personen (die also zu mehr als einer Gruppe gehören) mehrfach aufgenommen werden sollen (`@unionOfArrays.members`) oder nicht (`@distinctUnionOfArrays.members`). (Letzteres geht natürlich nicht, wenn die To-many-Relationship durch eine Instanz von `NSSet` gebildet wird. Warum? Denken Sie bitte darüber selbst nach. Dort gibt es also nur `@distinctUnionOfSets`.) Diese Operatoren benötigt man jedoch nur selten, weshalb ich auf die Dokumentation verweise.

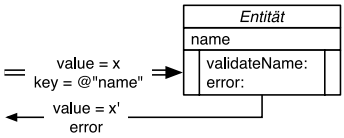
Key-Value-Coding



Key-Value-Coding erlaubt den schreibenden und lesenden Zugriff auf Eigenschaften einer Entität.

Key-Value-Coding (KVC) dient dazu, mittels Textschlüsseln Eigenschaften von Entitäten zu setzen oder zu lesen. Sie sind so etwas wie allgemeine Accessoren, die unabhängig von einer Klasse und deren Methoden verwendet werden können. Dabei kann aufgrund des Schlüssels (Key) ein Wert (Value) gelesen oder geschrieben werden.

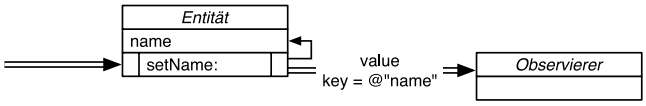
Key-Value-Validation



Darf ich? Key-ValueValidation liefert als Antwort »Ja«, »Nein« und »Jein: Zwar schon, aber etwas anders«.

Hiermit eng zusammen hängt Key-Value-Validation (KVV). Diese Technologie erlaubt die Überprüfung von Werten, die gespeichert werden sollen. Dabei kann der Empfänger den zu setzenden Wert akzeptieren, ablehnen oder anpassen.

Key-Value-Observation



Key-Value-Observing unterrichtet einen Observer über Änderungen der observierten Eigenschaft einer Entität.

Key-Value-Observing (KVO) ermöglicht es, die Veränderung von Eigenschaften einer Entität zu überwachen. Wird eine Eigenschaft verändert, so erhält der Überwacher eine sogenannte Observierungsnachricht.

Dies gilt unabhängig davon, ob der Wert nicht mittels Key-Value-Coding oder einem Standardsetter geschrieben werden soll. Also im obigen Beispiel auch bei einem Schreibzugriff mittels:

```
[observierter setName:@"x"];
```

Erst im Band 2 werden wir dazu übergehen, explizit Key-Value-Observation einzusetzen. Hier beschränken wir uns auf die Anwendung durch Cocoa-Bindings.

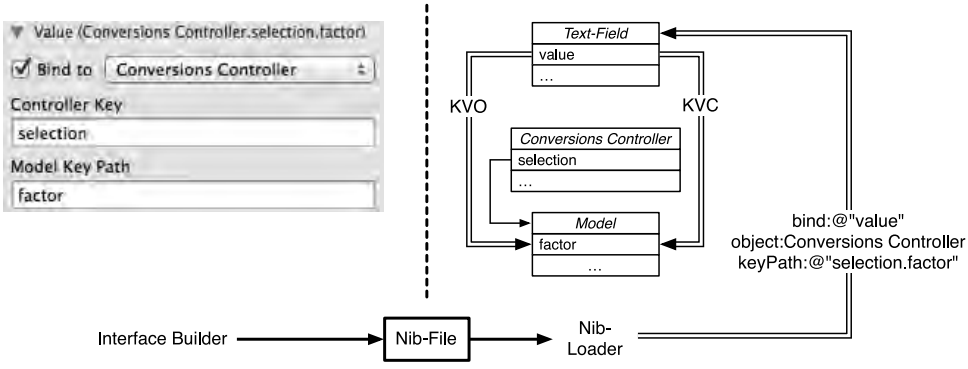
Cocoa-Bindings (Key-Value-Bindings)

Cocoa-Bindings kombinieren diese Technologien, um einheitliche Schnittstellen für die Synchronisation der Daten anzubieten. Es ist dabei ein weit verbreiteter Irrglaube, dass Cocoa-Bindings etwas Neues wären. Nein, ganz überwiegend sind sie eine reine, allerdings geschickte Kombination von Key-Value-Observing und Key-Value-Coding.

Um uns klar zu machen, wer die drei beteiligten Objekte sind, denken wir an das Faktor-Textfeld aus dem zweiten Kapitel. Dort hatten wir im Interface Builder (bestimmendes Objekt) gesagt, dass ein View (gebundenes Objekt) an die Eigenschaft `factor` des Controllers (observiertes Objekt) gebunden sein soll.

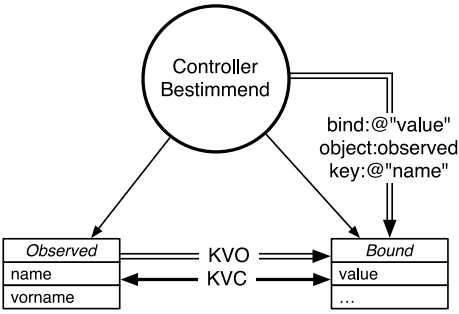
GRUNDLAGEN

Natürlich ist der Interface Builder kein Objekt unseres ablaufenden Programms. In Wahrheit verbirgt sich in unserem Programm als Klasse von Cocoa der Nib-Loader, der die im Interface Builder vorgenommenen Einstellungen in unserer Anwendung umsetzt. Daher ist es möglich, jede Einstellung, die wir im Interface Builder vornehmen, auch in unserem Code auszuführen. Etwas anderes macht der Nib-Loader ja nun auch nicht. Um etwa ein Binding zur Laufzeit zu setzen, existiert die Methode `–bind:toObject:withKeyPath:options:`.



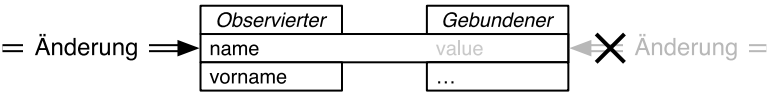
Der Nib-Loader setzt unsere Einstellungen zur Laufzeit um.

Es gibt also ein Objekt, welches die Bindungspartner festlegt und ein gebundenes Objekt (etwa View), welches eine Eigenschaft mit einem dritten Objekt (observiertes Objekt) synchronisiert. Daher verhält sich die Eigenschaft des gebundenen Objektes wie die Eigenschaften des observierten Objektes.



Dreierbande: Ein Controller bestimmt, welche Eigenschaft des gebundenen sich mit welcher Eigenschaft des Observierten synchropnisieren soll.

Ist daher das Binding erst einmal eingerichtet, so ist die Eigenschaft des Gebundenen nicht mehr als solche vorhanden. Sie spiegelt ja nur eine fremde Eigenschaft wider. Eine Änderung der gebundenen Eigenschaft ist daher unzulässig und führt zu unerwarteten Ergebnissen.



Die gebundene Eigenschaft verschwindet, da sie sich auf eine andere synchronisiert.

Diese Technologien dienen übrigens im Wesentlichen dazu, dass die Datensynchronisierung innerhalb des Programmes funktioniert. Sie dienen nicht dazu, irgendwelches Spezialverhalten zu implementieren. Es gibt daher nur wenige Ausnahmen, die auch noch immer weniger werden, welche dazu führen, dass man Bindings-Controller ableitet. Inzwischen sind die Möglichkeiten der existierenden Implementierung weit gesteckt. Hier beschränken wir uns also auf das Standardverhalten.

6.1.2 Key-Value-Coding

Sie haben bisher zwei Typen von Accessoren kennengelernt, und zwar den klassischen Stil mittels expliziter Nachrichten und die Dot-Notation von Objective-C 2. Ganz kurz noch einmal vorgestellt:

```
// Klassischer Accessoren
[person setFirstname:…];
… = [person firstname];

// Dot-Notation in Objective-C 2
person.firstname = …;
… = person.firstname;
```

Beide Varianten waren letztlich nur unterschiedliche Schreibweisen für dieselben Nachrichten. Sie teilten einen Nachteil: Die Eigenschaft `firstname` der Entität `person` ist bereits in unserem Sourecode fest enthalten. Sie wird so von dem Compiler übersetzt und lässt sich danach nicht mehr ändern. Das geht aber nicht, wenn wir eine solche Eigenschaft im Interface Builder setzen. Denn der Nib-Loader läuft erst nach der Übersetzung mit dem Programm. Er muss also die Möglichkeit haben, diese Zeilen nachträglich zu ändern, wenn unser Programm bereits gestartet ist und die entsprechende NIB-Datei geladen wird.

Funktionsweise

Oder versetzen Sie sich in die Lage des Programmierers, der ein Textfeld entwickelt hat. Der will den Wert aktualisieren. Aber wie kommt er daran? Er weiß ja nichts von unserem Modell, nichts von den von uns gewählten Eigenschaften und deren Namen. Wie soll sein Sourcecode aussehen:

```
value = [entity??? property???];
```

Gut, beim Adressaten könnte man ihm noch helfen. Man könnte den etwa einmal nach dem Laden des Nib setzen, sei es selbst oder sei es über den Nib-Loader. Aber bei der Eigenschaft hilft das nicht. Selbst wenn ich den Namen mitteile, kann ja nicht mehr nachträglich der Code im Textfeld geändert werden. Es läuft also darauf hinaus, dass ein automatisches Abholen des aktuellen Wertes durch das Textfeld mit den bisher besprochenen Mitteln nicht möglich ist.

Die Lösung dieses Problems erlaubt letztlich Key-Value-Coding, weil sie es ermöglicht, dass erst zur Laufzeit der Name der Eigenschaft bestimmt wird und daraus Nachrichen erzeugt werden. Dies sieht dann so aus:

```
// Key-Value-Coding-Accessoren
// statt [person set Firstname:…]
[person setValue:… forKey:@"firstname"]; // [person setFirstname:…]

// statt … = [person firstname]
… = [person valueForKey:@"firstname"]; // [person firstname]
```

»Moment, jetzt steht aber doch die Eigenschaft auch im Sourcecode«, höre ich Sie da sagen. Ja, das stimmt schon. Aber nicht mehr als fester Bestandteil, sondern als Parameter, und Parameter lassen sich bekanntlich ändern:

```
NSString* key = @"firstname";
[person setValue:… forKey:key];
… = [person setValueForKey:key];
```

Immer noch im Sourcecode? Gut, noch eine Stufe weiter:

```
NSString* key = … // Aus einer Datei lesen
[person setValue:… forKey:key];
… = [person setValueForKey:key];
```

Wir lesen jetzt also die NSString-Instanz aus einer Datei, also zur Laufzeit, und benutzen diese dann als Key. Sie ahnen es schon: Die Datei könnte eine Nib-Datei sein … Damit können wir nach der Übersetzung den Schlüssel ändern. Vielleicht noch ein anderes Beispiel zum Verständnis: Wir wollen eine Eigenschaft ermitteln, wobei wir erst zur Laufzeit bestimmen können, welche das sein soll. Sei es nur, weil der Benutzer sich das selbst auswählen kann, zum Beispiel bei der Konfiguration einer Spalte in einem Ausdruck. Statisch sieht das so aus:

```
key = …; // Irgendwoher, vllt User-Interface.
if( [key isEqualToString:@"firstname“] ) {
   value = [person firstname];
} else if( [value isEqualToString:@"lastname"] ) {
   value = [person lastname];
}
```

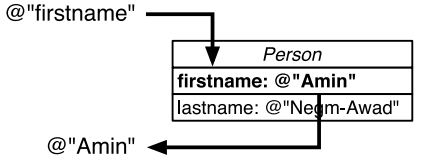
Wir haben also ein if zur Unterscheidung und darin bereits feste Methoden, die der Compiler übersetzt und auflöst. Anders mit Key-Value-Coding:

```
key = …; // Irgendwoher, vllt User-Interface.
value = [person valueForKey:key];
```

GRUNDLAGEN

Es wird also eine Indirektion hinzugefügt, wie das auch bei den Pointern war: Im Code steht nicht, welche Eigenschaft genommen werden soll, sondern, an welcher Stelle steht, welche Eigenschaft genommen werden soll. Statt firstname und lastname eben eine Variable, die firstname oder lastname als Inhalt hat.

Key-Value-Coding dient also vereinfacht gesagt dazu, erst zur Laufzeit zu bestimmen, welche Eigenschaft einer Entität benutzt werden soll.



Jede Eigenschaft lässt sich über einen Key spezifizieren.

Das erinnert sehr stark an die Dictionarys mit den Methoden `–setObject:forKey:` und `–objectForKey:` aus dem Kapitel 4, was den äußeren Zugriff angeht. Das Besondere ist

jedoch, dass eben die normalen Accessoren ihrer Klasse benutzt werden, so dass Sie diese überschreiben können. Auch bleibt ihre Klasse weiterhin ihre Klasse mit allen Möglichkeiten der Ableitung, Ansprache usw.

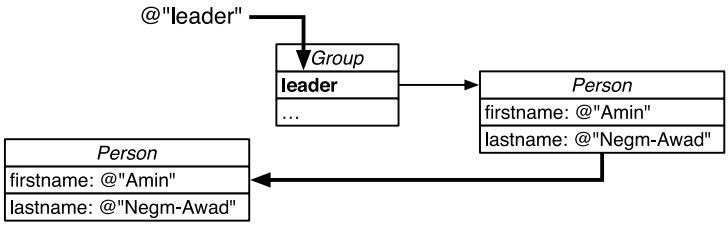
Einfache Accessoren (Getter und Setter)

Sie können also mit obigen Methoden auf verschiedene Eigenschaften zugreifen.

Nutzung

Dies funktioniert jedoch nicht nur, wenn die Eigenschaft ein Attribut ist, sondern auch bei Beziehungen zu einer Entität.

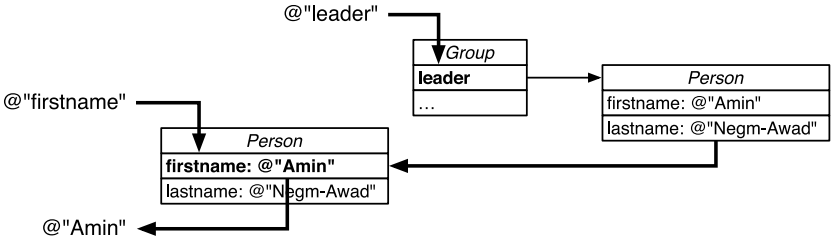
```
// Statt
// Person* person = [group leader];
Person* person = [group valueForKey:@"leader"];
```



Wenn die Eigenschaft eine Beziehung ist, erhalte ich eben die bezogene Instanz.

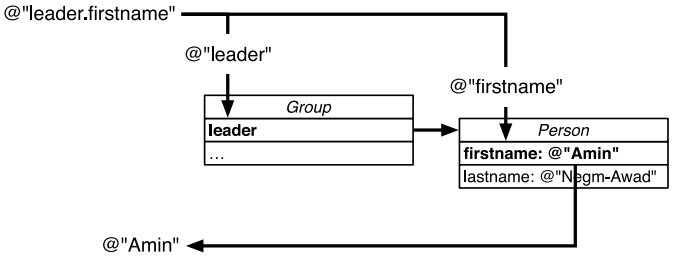
Auf das Ergebnis dieser Operation, also die Person, kann man wiederum Key-Value-Coding anwenden:

```
// Statt
// Person* person = [group leader];
Person* person = [group valueForKey:@"leader"];

// NSString* firstname = [person firstname];
NSString* firstname = [person valueForKey:@"firstname"];
```



Auf das Ergebnis einer Key-Value-Methode kann ich wieder einen Key anwenden.

Weil dies aber recht häufig vorkommt, bietet Key-Value-Coding gleich die Möglichkeit, einen ganzen Schlüsselpfad anzugeben:



Mehrere Keys ergeben einen Key-Path.

```
// Statt
// NSString* firstname = [[group leader] firstname];
NSString* firstname = [group valueForKeyPath:@"leader.firstname"];
```

Dabei existieren vier Key-Value-Accessormethoden:

- `-valueForKey:` liefert den unter dem angegebenen Namen gespeicherten Wert zurück.
- `-setValue:forKey:` setzt den als Value-Parameter angebenen Wert für die Eigenschaft mit dem als Key-Parameter angegebenen Namen.
- `-valueForKeyPath:` arbeitet wie -valueForKey:, es kann jedoch ein Schlüsselpfad angegeben werden.
- `-setValue:forKeyPath:` arbeitet wie `-setValue:forKey:`, wobei auch hier ein Schlüsselpfad angegeben werden kann.

Wenn ich also etwa den Vornamen eines Gruppenleiters setzen möchte, so lautet die entsprechende Anweisung:

```
[aGroup setValue:@"Hans" forKeyPath:@"leader.firstname"];
```

Implementierung

Dabei ersetzt Key-Value-Coding nicht unsere Accessoren, sondern benutzt diese. Es ist daher weiter erforderlich, diese zu programmieren oder (explizit oder implizit) zu synthetisieren. Man bezeichnet das als »KVC-Compliance« (KVC-Einhaltung). Der obige Code führt zu folgenden Anweisungen:

```
// [aGroup setValue:@"Hans" forKeyPath:@"leader.firstname"];
// wird:
id object = [aGroup leader];
[object setFirstname:@"Hans"];
```

AUFGEPASST

Wenn ich sage, dass die Accessoren weiterhin erforderlich sind, stimmt das nicht ganz: Wenn nämlich die Key-Value-Coding-Methoden keinen entsprechenden Accessor finden, versuchen sie, die Nachricht zu retten, indem sie unmittelbar auf die Instanzvariablen zugreifen, falls die Klasse dies mit der Methode `+accessInstanceVariablesDirectly` erlaubt. Von diesem Trick machen Sie bitte keinen Gebrauch. Key-Value-Coding wird übrigens auch noch einmal für Core Data besprochen.

Zusammenfassung:

- Anstelle eines Zugriffes mittels Accessornachrichten – explizit oder über Dot-Notation – können auch die KVC-Nachrichten verwendet werden.
- Dies setzt voraus, dass die Accessoren *-Eigenschaft* und (für schreibenden Zugriff) -set*Eigenschaft* für die entsprechenden Eigenschaften implementiert sind.
- Um einen Vorurteil vorzubeugen: Unerheblich ist, ob eine entsprechende Property deklariert wird oder ob die Methoden ausdrücklich im Header genannt sind. Dies interessiert ja nur den Compiler. Wir erzeugen die Nachrichten jetzt aber zur Laufzeit.

Ungeordnete To-many-Relationships (Sets)

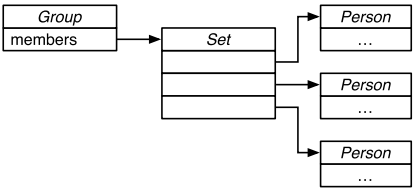
Dieses Key-Value-Coding, wie ich es bisher vorgestellt habe, ersetzt nur einfache Nachrichten der Art *Eigenschaft* als Getter und set*Eigenschaft*: als Setter. Dies gilt für alle Arten von Eigenschaften, also für Attribute, Master-Detail-Beziehungen und To-many-Beziehungen. (Vielleicht lesen Sie noch einmal den entsprechenden Abschnitt in Kapitel 3, um sich die Beziehungstypen in Erinnerung zu rufen.)

Problemstellung

Bei To-many-Beziehungen – wie etwa die Beziehung einer Gruppe zu deren Mitgliedern in unserem Firmenbeispiel – haben Setter einen Nachteil betreffend des Laufzeitverhaltens. Denn hier wird die Eigenschaft durch eine Collection repräsentiert. Damit etwa ein Mitglied der Gruppe hinzugefügt wird, wäre folgender Code erforderlich:

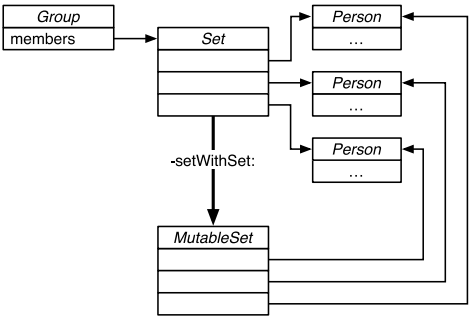
```
Group* aGroup =  … //
// Bisherige Mitglieder holen
NSSet* members = [aGroup valueForKey:@"members"];

// Veränderliche Collection erzeugen
NSMutableSet* members2 = [NSMutableSet setWithSet:members];

// Neues Mitglied hinzufuegen
[members2 addObject:[Person …]];

// und wieder speichern:
[aGroup setValue:members2 forKey:@"members"];
```

Gehen wir das einmal im Einzelnen durch: Zunächst haben wir eine Gruppe, die über ein Set drei Personen als Members hat.



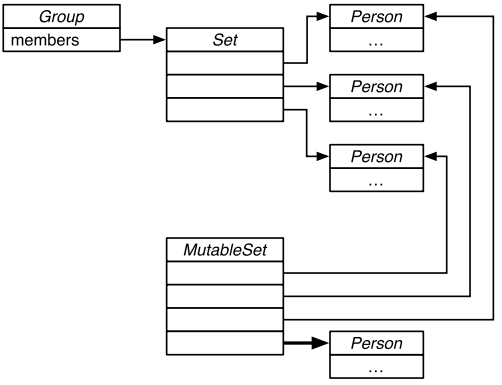
Die Ausgangssituation: Gruppe – Set – Personen

Dieses Set holen wir uns ab und erzeugen eine veränderliche Kopie mit `–setWithSet:`. Diese zeigt auf diesselben Personen wie das Ursprungsset, da bei der Kopie einer Collection die enthaltenen Mitglieder nicht mitkopiert werden.



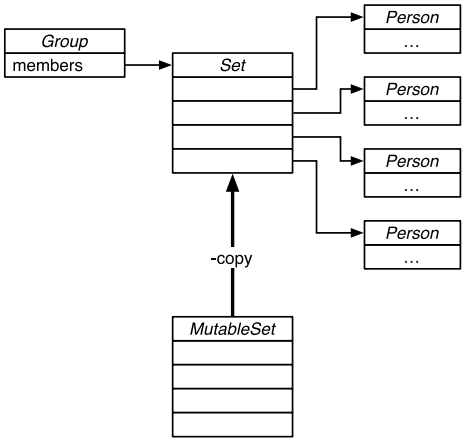
Wir erzeugen eine veränderliche Variante.

In das Mutable-Set fügen wir eine neue Person ein:



In das Mutable-Set können wir eine Person einfügen.

Letztlich wird durch die Nachricht setValue:forKey: am Ende des Codes das Setzen ausgeführt. Da es sich um eine Copy-Property handelte, wird hierbei wieder eine unveränderliche Kopie erzeugt.



Schließlich wird die Veränderung zurückkopiert.

Sie merken schon, dass das ziemlich umständlich ist. Es ist aber zudem nicht so einfach ersichtlich, welche Instanz in der Collection hinzukam. Sie sehen das natürlich sofort, wenn Sie die erste mit der letzten Graphik vergleichen. Nur diesen Vergleich hat aber die Entität, da sie von den Zwischenschritten nichts mitbekommt. Dies führt dazu, dass andere Technologien von Cocoa, wie das Key-Value-Observing, dies erst erneut ermitteln müssen, indem sie lokale Kopien des ursprünglichen Sets mit dem neuen vergleichen. Noch einmal von hinten durch die Brust ins Auge – auch wenn es geht!

Nutzung

Aus diesem Grunde gibt es besondere Methoden, die dem Anwender von Key-Value-Coding das Leben erleichtern und gleichzeitig für eine genauere Änderung der Collection sorgen. Dies sind `–mutableSetValueForKey:` und `–mutableSetValueForKeyPath:` für eine Beziehung, die über eine Instanz der Klasse `NSSet` modelliert wurde:

```
Group* aGroup =  … //

// Bisherige Mitglieder holen
NSMutableSet* members = [aGroup mutableSetValueForKey:@"members"];

// Neues Mitglied hinzufuegen
[members addObject:[Person …]];
```

Der Trick liegt darin, dass wir ein ganz spezielles Set bekommen, welches sich um das weitere Key-Value-Coding kümmert.

Implementierung von Setter und Getter

Sollen weiterhin Setter und Getter vorhanden sein – dazu später noch –, so taucht das Problem auf, dass diese nicht synthetisiert erzeugt werden können. Klarer wird das alles an einem Beispiel. Wir haben wieder eine Gruppenklasse mit einer To-many-Relationship zu Personen, die wir `members` nennen. Sie kennen das aus dem Abschnitt über Automatic Reference Counting. Schauen wir uns das noch einmal an:

```
@interface Group : NSObject
@property (copy)   NSString *name;
@property (strong) Person   *leader;
@property (copy)   NSSet    *members;
@end
```

Die Accessoren und die Instanzvariable wurden bisher einfach aus der Property synthetisiert. Da ist einiges zu ändern:

- Die Instanzvariable muss jetzt intern den Typen `NSMutableSet*` haben, damit wir leicht einzelne Objekte einfügen und löschen können.
- Die Property darf aber nicht diesen Typen haben, sondern weiterhin `NSSet*`. Sonst könnte ein Nutzer der Klasse sich das Set abholen und auf gewöhnliche Weise, ohne, dass wir es merken, Änderungen vornehmen: Die Todsünde der objekt-orientierten Programmierung.
- `copy` führt als Option der Property dazu, dass der synthetisierte Setter `copy` sendet. Dies erzeugt aber von einer veränderlichen Instanz wie die von `NSMutableSet` eine unveränderliche Kopie, also eine Instanz von `NSSet`. Gespeichert wäre daher in der Instanzvariablen ein unveränderliches Set, in das wir wieder nicht bequem Objekte einfügen usw. könnten.

So wie bisher geht es also nicht mit der Property und vor allem der Synthetisierung. Zunächst einmal müssen wir die Instanzvariable selbst erzeugen, anstatt sie zu synthetisieren. Die Implementierung von `Group` ändert sich also:

```
@implementation Group {
   NSMutableSet *_members;
}
```

Außerdem müssen wir was mit den Accessoren tun. Möglich ist es, die Eigenschaft auf `readonly` zu setzen und nur den Getter synthetisieren zu lassen. Zum einen ist es zulässig – wenngleich nicht dokumentiert –, dass eine Synthetisierung von einem Typen (`NSSet`) mit einem Subtypen (`NSMutableSet`) verbunden wird. Zum anderen ändert der Getter ja nicht den Wert der Instanzvariable, so dass er auch nichts kaputt machen kann. Das sähe dann so aus:

```
// Property hat readonly-Attribute
@synthesize members = _members; // Explizit
```

GRUNDLAGEN

Der Getter würde eine Instanz von `NSMutableSet` zurückliefern, da er kein `copy` versendet. Der Nutzer der Klasse könnte den Rückgabewert explizit auf `NSMutableSet*` casten und dann wiederum ohne Mitteilung an die Instanz ändern. Hier würde also ein Programmierer explizit die Kapselung der Objektorientierung umgehen. Ich meine nicht, dass man ihn davor schützen muss, weil das kein Versehen, sondern kriminelle Energie ist: Selbst Schuld!

Will man einen Setter haben, besteht die Lösung darin, eigene Accessoren zu bauen. Diese sehen dann etwa so aus:

```
- (NSSet*)members { return _members; }
- (void)setMembers:(NSSet*)members { _members = [members mutableCopy]; }
```

Beachten Sie hier das `mutableCopy`, um intern eine veränderliche Instanz zu speichern. Man kann freilich auch ganz darauf verzichten, Standardaccessoren zu haben oder die ohne jegliches `@property` nur intern programmieren. So häufig müssen gar nicht To-many-Relationships als Ganzes gesetzt werden.

Implementierung von To-many-Accessoren

Aber auch hier ist noch nicht das Optimum erzielt. `–mutableSetValueForKey: pp.` könnten ja wieder nur die Getter und Setter benutzen. Damit wäre das Problem, dass die Collection erst umständlich kopiert und dann zurückgeschrieben werden muss, nur verlagert. Aber wir können den Methoden helfen, ihre Arbeit performant zu erledigen. Dazu müssen wir in der entsprechenden Entität zusätzlich spezielle Key-Value-Coding-Methoden implementieren. Für ein Set lauten diese:

- `–countOf`*`Eigenschaft`* liefert als Ergebnis die Anzahl der verwiesenen Objekte.
- `–enumeratorOf`*`Eigenschaft`* muss einen Enumerator für die Collection zurückgeben.
- `–memberOf`*`Eigenschaft`*: erhält als Parameter ein Objekt, welches mit den gespeicherten verglichen wird. Ist eine bereits gespeicherte Instanz inhaltlich gleich, so muss dieses zurückgegeben werden, ansonsten nil.
- `–add`*`Eigenschaft`*`Object`: soll das übergebene Objekt der Collection hinzufügen, die für die Eigenschaft *Eigenschaft* steht. In unserem Falle also etwa -addMembersObject:.
- `–add`*`Eigenschaft`*`:` fügt die Mitglieder eines übergebenen Sets der Eigenschaft hinzu.
- `–remove`*`Eigenschaft`*`Object:` entfernt entsprechend wieder ein Objekt. In unserer Klasse Group hieße also die Methode -remove*Members*Object:.
- `–remove`*`Eigenschaft`*: entfernt die Mitglieder eines übergebenen Sets.
- Falls implementiert, verwendet das spezielle Set auch eine Methode `–intersect`*`Eigenschaft`*`:`, wobei als Ergebnis diejenigen Elemente gespeichert werden müssen, die sowohl in der bisherigen Menge gespeichert waren als auch in der übergebenen (Schnittmenge).

Beachten Sie bitte, dass eine solche To-many-Relationship als Namen einen Plural trägt, etwa `persons`. Daher heißen die Methoden beispielsweise `–addPersonsObject:` und `–addPersons:`. Nehmen wir eine Beispielimplementierung vor. Xcode gibt uns die genauen Namen mittels Code-Completion vor, wenn wir nach einem Bindestrich mit dem Namen anfangen, also etwa *-cou* getippt wurde. Das vereinfacht die Arbeit ein bisschen:

```
-(NSUInteger)countOfMembers { return [_members count]; }
-(NSEnumerator *)enumeratorOfMembers { return [_members objectEnumerator]; }
-(Person*)memberOfMembers:(Person*)object { return [_members member:object]; }

-(void)addMembers:(NSSet*)objects { [_members unionSet:objects]; }
-(void)addMembersObject:(Person*)object { [_members addObject:object]; }
-(void)removeMembers:(NSSet*)objects { [_members minusSet:objects]; }
-(void)removeMembersObject:(Person*)object { [_members removeObject:object]; }
-(void)intersectMembers:(NSSet*)objects { [_members intersectSet:objects]; }
```

Es reicht aus, eine der beiden add…-Methoden und eine der beiden remove…-Methoden zu implementieren. Auch die intersect…-Methode ist optional. Sie sehen schon, dass das derart langweilig ist, dass ich die Methoden einzeilig programmiert habe. Jetzt hat unsere Klasse Group also passgenaue To-many-Accessoren, die für besseres Laufzeitverhalten sorgen. Wenn wir diese auch aus unserem eigenen Code heraus benutzen wollen, können wir die Methoden natürlich im Header bekannt machen. Aber es geht eben auch über `–mutableSetValueForKey:`.

Diese Methoden setzen allerdings voraus, dass die Instanzvariable bereits eine leere Instanz hat.

```
- (id)init
{
   self = [super init];
   if( self ) {
      _members = [NSMutableSet set];
   }
   return self;
}
```

Wenn man weiterhin einen Setter hat, kann man den freilich benutzen. Eine andere Variante ist es, in jeder Accessormethode für Sets nachzufragen, ob bereits ein Set vorhanden ist und dies gegebenenfalls nachzuholen (Lazy-Evaluation). Exemplarisch für eine Methode:

```
- (void)addMembers:(NSSet*)values
{ _if( members == nil ) {
      _members = [NSMutableSet set];
   }
   [_members unionSet:values];
}
```

Geordnete To-many-Relationships (Arrays, Ordered-Sets)

Dasselbe Problem stellt sich, wenn eine Beziehung nicht mit einem Mutable-Set, sondern mit einem Mutable-Array implementiert wurde.

Nutzung

Der Zugriff auf eine solche Eigenschaft wird dann mit `–mutableArrayForKey:` und `–mutableArrayForKeyPath:` erledigt. Es wird ein Mutable-Array zurückgegeben, welches die üblichen Methoden versteht und automatisch an die ursprüngliche Eigenschaft weiterleitet. Dasselbe wie bei Sets, nur eben Array-Methoden.

Für Ordered-Sets, die ja auch indiziert sind, gilt Entsprechendes mit `–mutableOrderedSetValueForKey:` bzw. `–mutableOrderedSetValueForKeyPath:` .

Implementierung von Setter und Getter

Es gilt nichts Abweichendes zu Sets.

Implementierung von To-many-Methoden

Da der Zugriff auf ein Array anders erfolgt als auf ein (ungeordnetes) Set, müssen wir entsprechend zur Optimierung andere Methoden implementieren. Nehmen wir wieder ein Beispiel:

```
@implementation Group {
   NSMutableArray* members;
}
```

Dafür ergibt sich:

```
- (NSUInteger)countOfMembers
   { return [_members count]; }

- (id)objectInMembersAtIndex:(NSUInteger)index
   { return [_members objectAtIndex:index]; }

- (void)getMembers:(__unsafe_unretained id*)objsPtr range:(NSRange)range
   { [_members getObjects:objsPtr range:range]; }

- (void)insertObject:(id)obj inMembersAtIndex:(NSUInteger)index
   { [_members insertObject:obj atIndex:index]; }

- (void)insertMembers:(NSArray*)objs AtIndexex:(NSIndexSet*)indexes
   { [_members insertObjects:objs atIndexes:indexes]; }

- (void)removeObjectFromMembersAtIndex:(NSUInteger)index
   { [_members removeObjectAtIndex:index]; }

- (void)removeMembersAtIndexes:(NSIndexSet*)indexes
   { [_members removeObjectsAtIndexes:indexes]; }
```

```
- (void)replaceObjectInMembersAtIndex:(NSUInteger)index withObject:(id)obj
   { [_members replaceObjectAtIndex:index withObject:obj]; }

- (void)replaceMembersAtIndexes:(NSIndexSet*)indexes withMembers:(NSArray*)objs
   { [_members replaceObjectsAtIndexes:indexes withObjects:objs]; }
```

Sie sehen also, dass dies gar nicht so schwierig ist, vielmehr in der Standardimplementierung nur Nachrichten an das zugrunde liegende Array weitergeleitet werden. Verpflichtend ist die Implementierung der Getter und von jeweils einer Methode aus den Gruppen -add…, -remove… und -replace…

Bei Ordered-Sets werden freilich als Parameter Instanzen von `NSOrderedSet` verwendet.

Wie gesagt: Dies alles ist nicht erforderlich. Sie können weiterhin wie bisher mit unveränderlichen Collections arbeiten und dann synthetisieren, oder zu Fuß nur die Standard-Accessoren implementieren. Es geht hier um Optimierung.

Zusammenfassung:

- Der Zugriff auf Eigenschaften kann mittels KVC-Nachrichten erfolgen: `valueForKey:`, `setValue:forKey:`, `mutableSetValueForKey:`, `mutableArrayValueForKey:` und `mutableOrderedSetValueForKey:` sowie deren Pendants mit KeyPath anstelle von Key.
- Bei Attributen müssen Setter und Getter implementiert werden.
- Diese reichen auch bei To-many-Relationships aus, haben aber einen bedeutenden Performancenachteil. Es sollten daher die entsprechenden Accessormethoden für To-many-Relationships implementiert werden.

HILFE

Wenn Sie den nächsten Abschnitt über Key-Value-Observing gelesen haben, verstehen Sie auch ein weiteres Tutorial, welches sich auf meiner Webseite befindet. Dieses geht auf die Probleme bei einem gedankenlosen Umgang mit mutablen Collections genauer ein.

Fehlermethoden

Bei `–valueForKey:` und `–setValue:forKey:` kann es natürlich passieren, dass der angegebene Schlüssel gar nicht bei der Instanz vorhanden ist, die diese Nachricht bekam. In diesem Falle wird versucht, die Nachricht zu retten, indem die Methoden `–valueForUndefinedKey:` bzw. `–setValue:forUndefinedKey:` in der Empfängerinstanz aufgerufen werden. Die Standardimplementierungen in `NSObject` erzeugen dann einfach einen Fehler. Wir können diese Methoden aber überschreiben. Das ist zuweilen praktisch, wenn wir ein Zwischen- oder Stellvertreterobjekt (Proxy) haben, welches selbst die Eigenschaft nicht implementiert, aber auf jemanden Rückgriff nehmen kann, der das tut. Eine Implementierung dieser Funktionalität sieht etwa so aus:

```
- (id)valueForUndefinedKey:(NSString*)key
{
   return [[self recourse] valueForKey:key];
}
- (void)setValue:(id)value forUndefinedKey:(NSString*)key
{
   [[self recourse] setValue:value forKey:key];
}
```

Eine weitere nützliche Methode stellt `–setNilValueForKey:` dar, die aufgerufen wird, wenn für einen Skalar wie einen Integer `nil` gesetzt werden soll. Wie bereits bei den Collections erläutert, existiert ein semantischer Unterschied zwischen Nichts (`nil`) und dem Wert 0.

6.1.3 Key-Value-Validation

Cocoa deklariert als informelles Protokoll (Kategorie von `NSObject`) ebenso die Methoden `–validateValue:forKey:error:` und `–validateValue:forKeyPath:error:`, die ihrerseits eine Methode `–validateEigenschaft:error:` für die Überprüfung in der jeweiligen Klasse aufrufen.

Die entsprechende Validierungsmethode bekommt einen Vorschlag über ihren ersten Parameter und liefert bis zu zwei Werte zurück. Dabei sind drei Fälle zu unterscheiden:

- Über den ersten Parameter erhält die Methode den Vorschlag, insbesondere also eine Benutzereingabe. Stellt diese Benutzereingabe einen zulässigen Wert dar, so wird einfach gar nichts unternommen und `YES` als Returnwert zurückgegeben.
- Ist der erste Parameter zwar nicht akzeptabel, lässt sich aber aus ihm ein akzeptabler Wert ableiten, so wird dieser erzeugt und über die Parameterliste (Vorsicht: Zeiger-Zeiger!) zurückgegeben. Auch in diesem Falle muss `YES` der Returnwert sein.
- Ist der erste Parameter nicht akzeptabel und lässt sich auch keiner erzeugen, so wird über den Error-Parameter (Vorsicht: Zeiger-Zeiger!) ein Error-Objekt zurückgeliefert. Der Returnwert muss hier auf `NO` lauten.

Stellen wir uns eine Eigenschaft `lastname` vor, die nur Namen akzeptiert, die aus höchstens vier Buchstaben bestehen. Eine entsprechende Validierungsmethode sähe so aus:

```
- (BOOL)validateLastname:(id*)value error:(NSError**)error
{
   // nil ist als Wert immer erlaubt.
   // Beachte aber den Zeiger-Zeiger!
   if( *value == nil ) {
      return YES; // alles okay
   }
```

```
   // Ein Wert mit bis zu 4 Buchstaben ist erlaubt:
   if( [(*value) length] <= 4 ) {
      return YES;
   }
   // Einen Wert mit mehr als 4 Buchstaben kann man anpassen:
   *value = [(*value) substringToIndex:4];
   return YES;
}
```

Soll indessen der Name mindestens vier Buchstaben enthalten, ist es natürlich nicht möglich, eine falsche Eingabe zu retten. Entsprechender Code:

```
- (BOOL)validateLastname:(id*)value error:(NSError**)error
{
   // nil ist als Wert nicht erlaubt.
   if( *value == nil ) {
      *error = [NSError errorWithDomain:@"reverse DNS"
                                   code:1
                               userInfo:@"Name zu kurz"];
      return NO; // Fehler!
   }
   // Ein Wert mit weniger als 4 Buchstaben ist nicht erlaubt:
   if( [(*value) length] < 4 ) {
      *error = [NSError errorWithDomain:@"reverse DNS"
                                   code:1
                               userInfo:@"Name zu kurz"];
      return NO; // Fehler!
   }
   return YES;
}
```

Die Domain müssen Sie freilich nach dem bereits beschriebenen System der Reverse-Domains umbenennen.

6.1.4 Key-Value-Observing

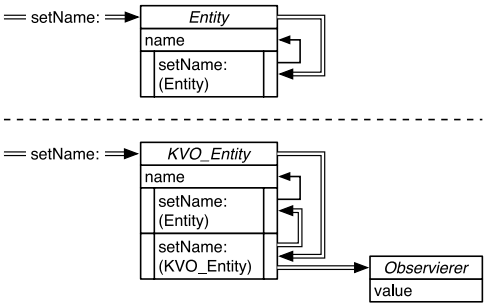
Key-Value-Observing sorgt dafür, dass die Änderung einer Eigenschaft zu einer Mitteilung an einen Observierer führt. Es ist damit die Grundlage von Bindings. Wir werden uns damit und mit Bindings hier nur strukturell beschäftigen, da Key-Value-Observing einiges Fortgeschrittenenwissen voraussetzt – ebenso wie die eigene Anwendung außerhalb von Bindings.

Sie haben das wörtlich zu nehmen: Key-Value-Observing überwacht eine Eigenschaft, nicht eine Instanzvariable! Dies bedeutet, dass eine spezielle Observierungsklasse mit

überschriebenen Accessoren zur Laufzeit erzeugt wird und die observierte Instanz dieser Klasse zugewiesen wird. Wird an irgendeiner Stelle im Programm eine Nachricht an die eigentliche Entität geschickt, so wird diese Nachricht von dem überschriebenen Setter der Observierungsklasse abgefangen und eine Observierungsnachricht ausgelöst.

GRUNDLAGEN

Den ein oder anderen erinnert das vielleicht an AOP. Der Unterschied besteht darin, dass bei KVO die Einrichtung der Observierung dezidiert für bestimmte Instanzen erfolgt.



Durch eine Subklasse wird zusätzlicher Code eingeschleust.

Alle nicht überschriebenen Methoden werden freilich wieder in der Basisklasse ausgeführt. Für Zweifler: Sogar die Methode `–class` gibt die Basisklasse zurück, damit sich die KVO-Klasse verbirgt.

Damit das funktioniert, muss die Subklasse natürlich wissen, welche Nachrichten potenziell die Entität verändern. Und hier gilt eine Vereinbarung, dass nur die im Abschnitt Key-Value-Coding genannten Methoden dies tun. Dies ist auch der Hintergrund dafür, dass Sie stets eine der Standardaccessoren oder der erweiterten Methoden für To-many-Relationships verwenden müssen, um eine Eigenschaft zu verändern. Alles andere führt dazu, dass die Observierung nicht mehr funktioniert und das Programm fehlerhaft abläuft.

Also, bevor Sie tiefer in die Materie eingestiegen sind, können Sie es sich einfach machen:

- Bilden Sie alle Attribute nach außen mit Immutable-Containern ab, also zum Beispiel mit `NSString` anstelle von `NSMutableString`.
- Bilden Sie alle To-many-Relationships nach außen mit unveränderlichen Collections ab, also zum Beispiel mit `NSSet` anstelle von `NSMutableSet`.
- Verwenden Sie zur Veränderung von sämtlichen Eigenschaften stets nur die Standardaccessoren (ruhig auch in der Form der Dot-Notation) oder die KVC-Methoden. Diese Regel nennt man KVO-Compliance.

TIPP

Wenn Sie mal in einer ruhigen Minute die Dokumentation zu Key-Value-Coding und Key-Value-Observing durcharbeiten, werden Sie möglicherweise sehen, dass für Eigenschaften, die einen `BOOL` als Datentypen haben, auch andere Bezeichnungen für den Getter erlaubt sind. Ich würde Ihnen empfehlen, hiervon keinen Gebrauch zu machen, da dies nur eine weitere Ausnahme in Ihren Sourcecode einführt.

6.1.5 Bindings

Sie haben bereits Bindings verwendet. Dabei sind wir allerdings sehr simpel vorgegangen. Vermutlich bemerkten Sie schon, dass man im Inspector des Interface Builders viel mehr Einstellungen vornehmen konnte. Hierauf will ich eingehen.

Bindbare Eigenschaften

HILFE

Bitte laden Sie das Projekt Converter-28 von der Webseite herunter. Es handelt sich um ein wieder auf die Kernfunktionalität abgespecktes Converter-Projekt.

Jedes Objekt, insbesondere solche der View-Schicht, können verschiedene Eigenschaften zur Bindung anbieten. Öffnen Sie das heruntergeladene Projekt. Ziehen Sie jetzt einen Button in das Dokumentenfenster, und beschriften Sie ihn mit *Entfernen*.

Wie bereits in Kapitel 2 vorgenommen, setzen Sie seine Action auf den Arraycontroller *Conversions Controller* und wählen dort die Methode *remove:* aus. Wenn Sie jetzt das Programm starten und testen, sollten wiederum über die Toolbar hinzugefügte Einträge gelöscht werden können. Ja, ja, ich weiß, soweit waren wir schon. Aber es geht hier um etwas anderes:

Löschen Sie über die Toolbar oder den Button alle Einträge. Es ist jetzt natürlich keiner selektiert. Ich hatte Sie bereits darauf hingewiesen, dass sich das Entfernen-Item in der Toolbar automatisch grau darstellt und nicht mehr zu bedienen ist. Das ist auch richtig: Wenn kein Eintrag vorhanden ist, kann auch nichts gelöscht werden. Unser Button ist aber noch bedienbar. Glücklicherweise führt seine unsinnige Betätigung allerdings nicht zum Absturz des Programms, sondern lediglich dazu, dass einfach nichts passiert.

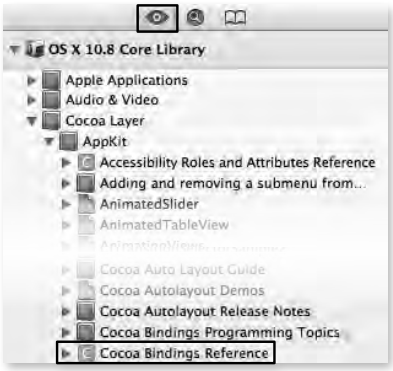
Es stellt sich also die Frage, wie wir dafür sorgen können, dass auch unser Button ausgegraut wird, wenn kein Eintrag in der Tabelle selektiert ist. Der Trick besteht wiederum in den Bindings.

Im Interface Builder selektieren Sie diesen Button und wechseln auf den Bindings-Inspector. Sie sehen dort eine ganze Reihe verknüpfbarer Eigenschaften.

Zahlreiche Eigenschaften eines Buttons können gebunden werden.

TIPP

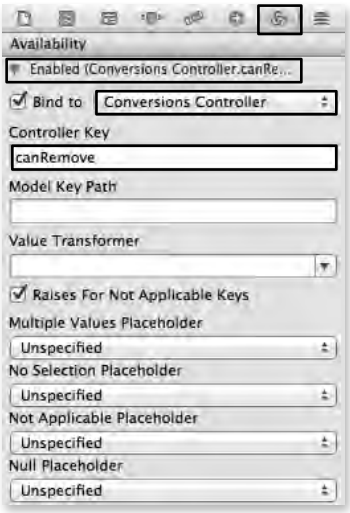
Natürlich kann ich hier bei Weitem nicht sämtliche bindbaren Eigenschaften besprechen. Das ist wieder das Problem mit dem Telefonbuch von New York. Sie können aber eine Dokumentation zu den einzelnen Bindings abrufen, wenn diese auch sehr versteckt ist: Hierzu öffnen Sie die Dokumentation in Xcode und wechseln im Menü *Editor* mit *Explore Documentation* auf die hierarchische Ansicht der Dokumentation (oder klicken Sie in der Leiste oberhalb der Treffer auf das linke Symbol). Dort müssen Sie den Disclosure *OS X 10.8 Core Library* | *Cocoa Layer* | *AppKit* | *Cocoa Bindings Reference* folgen. Weil dies umständlich zu erreichen ist, setzen Sie bitte gleich mit *Editor* | *Add Bookmark* ein Lesezeichen darauf. Lesezeichen erreichen Sie über *Editor* | *Documentation Bookmarks* bei geöffneter Dokumentation (oder über das rechte Symbol in der Leiste). Wechseln Sie jetzt aber wieder im Menü *Editor* auf *Search Documentation*, weil man das doch am häufigsten braucht. (Das wäre dann das mittlere Symbol.)



Gut gesucht ist halb gewusst: der Ort der Bindings-Referenz.

Wir wollen ja erzielen, dass der Button nicht anzuklicken ist, wenn kein Eintrag im Tableview ausgewählt ist. Deshalb müssen wir die bindbare Eigenschaft *Enabled* des Buttons verwenden. Klicken Sie auf den entsprechenden Disclosure, um das Binding zu öffnen. Schauen Sie sich zum Verständnis vielleicht noch einmal die Graphik vom Anfang an: Wir haben jetzt die Eigenschaft *Enabled* der Instanz Button, die sich synchron verhalten soll.

Synchron zu was? Über unsere Selektierung weiß ja der Arraycontroller Bescheid, den wir *Conversions* genannt hatten. Und deshalb binden wir den an. Und der hat wiederum eine observierbare Eigenschaft `canRemove`. Wenn also der Conversions-Controller uns sagt, dass ein Eintrag bei ihm gelöscht werden kann, so können wir unseren Button auf *Enabled* setzen. Deshalb binden wir das genau so:



Die Bindung des Enabled-Status an die canRemove-Eigenschaft

Einen Model-Key-Path müssen wir nicht eingeben, da wir uns ja mit einer Eigenschaft des Arraycontrollers selbst synchronisieren wollen. Also, noch einmal in einem Satz: Der Button ist enabled, wenn der Arraycontroller einen Eintrag löschen kann. Oder anders: Die gebundene Eigenschaft `enabled` des Buttons verhält sich synchron zu der observierbaren Eigenschaft `canRemove` des Controllers.

Bitte starten Sie jetzt das Programm, um dies zu testen. Einfach Einträge einfügen und löschen, bis keiner mehr da ist.

HILFE

Sie können das Projekt in diesem Zustand als Projekt Converter-29 von der Webseite herunterladen.

Gut, dass war jetzt einfach ein Beispiel, um Ihnen ein anderes Binding als dieses ewige Value zu demonstrieren. Entfernen Sie den Button wieder.

GRUNDLAGEN

Das Gegenstück der bindbaren Eigenschaften, also das, woran gebunden wird, heißt demnach eine observierbare Eigenschaft. Eine Eigenschaft ist observierbar, wenn sie KVC-compliant ist. Wie Sie für KVC-Compliance sorgen, habe ich ja ausführlich erläutert. Umgekehrt eine bindbare Eigenschaft (also ein Binding) selbst anzubieten, ist alles andere als lustig und bleibt Band 2 vorbehalten. Wir benötigen dies in der Regel auch nicht, da die Standardviews von Cocoa ausreichend Bindings anbieten.

Bindings-Optionen

Ich will noch die wichtigsten Optionen eines Bindings besprechen, die man im alltäglichen Gebrauch benötigt:

Bindings-Placeholder

Zu jedem Binding können Sie verschiedene Parameter angeben. Auch dies haben Sie vermutlich bereits bemerkt. Wählen Sie das Textfeld für den Umrechnungsfaktor an, und öffnen Sie im Bindings-Pane des Inspectors das *Value*-Binding. Am Ende sehen Sie eine Liste von Platzhaltern (Placeholder). Geben Sie im Feld unterhalb von *No Selection Placeholder* den Wert 0 ein. Dieser wird angezeigt, wenn Sie keine Auswahl im Tableview getroffen haben und damit der Arraycontroller keine Selektion kennt. Wenn Sie das Programm starten und sich noch kein Eintrag im Tableview befindet (also auch keiner ausgewählt ist), sehen Sie das Ergebnis.

Ich denke, dass sich die weiteren Placeholder von selbst erklären. Sie sind auch in der Bindings-Referenz erläutert. Nur eines: Der Multi-Selection-Placeholder ist dienlich, wenn man die mehrfache Selektion in einem Tableview (und damit dem Controller) erlauben würde. Hiermit in Zusammenhang steht die gleich besprochene Einstellmöglichkeit *Always use Mutliple Values Marker.*

Conditionally Sets …

Mit diesen Checkboxen können Sie bestimmen, ob die Eigenschaften `editable`, `hidden` usw. automatisch gesetzt werden sollen, wenn das Binding nicht zu einem gültigen Wert führt. Wenn Sie etwa für das mittlere Textfeld *Conditionally Sets Hidden* anklicken, verschwindet es automatisch, falls kein gültiger Wert ausgewählt ist.

Continuously Updated Values und Validates immediately

Auch Bindings übernehmen einen Wert erst, wenn die [Eingabe]-Taste gedrückt oder das Feld verlassen wird. Dies bedeutet, dass eine Änderung in dem Textfeld für den Umrechnungsfaktor erst dann in unserem Model gespeichert wird. Manchmal will man jedoch, dass bereits während der Eingabe nach jedem Tastendruck der entsprechende Wert im Model gesetzt wird. Dann aktivieren Sie diese Option.

Entsprechendes gilt für die Validierung des eingegebenen Wertes bei *Validates immediately*, wobei die Validierung bei ausgeschalteter Option bis zum Speichern ganz ausbleibt.

Value-Transformer

Vermutlich ist Ihnen bereits aufgefallen, dass man auch einen sogenannten Value-Transformer angeben kann. Hierbei handelt es sich um gesonderte Instanzen, die zur konstanten Umrechnung von Werten verwendet werden können. Dabei existiert immer eine Instanz, die einen Eingangswert aus der observierbaren Eigenschaft bekommt und daraus einen Ausgangswert für das Binding herstellt. Dies ist aber eine reine 1-zu-1-Umrechnung, so dass nicht mehrere observierbare Eigenschaften kombiniert werden können. Manche Value-Transformer können dabei auch die Rückrechnung vornehmen, also bei Eingabe der gebundenen Eigenschaft im User-Interface auch wieder die observierte Eigenschaft im Model setzen.

Getreu dem Motto dieses ersten Bandes »Lerne Cocoa richtig anwenden« will ich hier die wichtigsten vom System bereitgestellten Value-Transformer kurz benennen. Es sei aber angemerkt, dass man recht leicht eigene Transformer programmieren kann. Die Dokumentation zu der Klasse `NSValueTransformer` enthält vollständigen Beispielcode, der kein Hexenwerk ist.

NSNegateBoolean

Dieser Value-Transformer dient für boolsche Werte und verneint diese immer. Er kann für den Zustand (etwa die Enabled-Eigenschaft) von Buttons hilfreich sein, wenn die observierte Eigenschaft »genau verkehrt herum« ist. Außerdem lässt sich hiermit ein Status für die Benutzeroberfläche umdrehen.

Der Negate-Boolean-Value-Transformer beherrscht auch die umgekehrte Transformation.

NSIsNil, NSIsNotNil

Diese beiden Transformer erzeugen einen boolschen Wahrheitswert in Abhängigkeit davon, ob die observierte Eigenschaft `nil` ist. Auch hiermit lassen sich häufig Zustände im User-Interface darstellen. Allerdings existiert keine umgekehrte Transformation, so dass sie sich nicht zur Eingabe von Werten eignen.

NSUnarchiveFromData, NSKeyedUnarchiveFromData

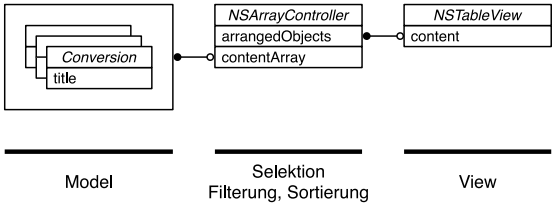
Diese beiden Transformer erwarten als observierbare Eigenschaft eine Instanz der Klasse `NSData` und wandeln die enthaltenen Daten in das gewünschte Objekt um. Mit diesem Value-Transformer und dem dafür erforderlichen Protokoll `NSCoding` beschäftigen wir uns noch im Kapitel für die Model-Schicht. Der Transformer erlaubt auch die Rückumwandlung von der gebundenen zu der observierten Eigenschaft.

6.1.6 Der Arraycontroller

Ich möchte zwei Controller, die in der praktischen Anwendung sehr wichtig sind, noch besprechen. Ich beginne dabei hier mit dem Arraycontroller sozusagen prototypisch. Die anderen Controller funktionieren entsprechend.

In der Regel wollen wir ja ein View an den Controller und den Controller an das Model binden. Das hatten wir auch mit unseren Conversions so gemacht, die im Model gespeichert wurden und in einem View erschienen. Dazwischen liegen Bindings-Controller wie der Arraycontroller. Sie haben daher zwei Seiten:

- Zum Model hin lassen sie sich selbst binden. Hier sind die verknüpfbaren Eigenschaften `contentSet` und `contentArray` wichtig. Werden diese – so wie wir es bereits getan haben – an das Model gebunden, so laufen sie also synchron zum Model. Dies bedeutet im Wesentlichen, dass die Eigenschaft `arrangedObjects` des Controllers die Elemente enthält, die sich im `contentSet` bzw. `contentArray` befinden.
- Zu dem View hin sind Controller die observierten Objekte. Sie bieten wichtige Eigenschaften an, die man observieren kann, insbesondere `arrangedObjects` als Ansammlung aller Objekte und `selection` als das soeben ausgewählte Objekt.
- Da es sich um verschiedene Eigenschaften handelt, kann dazwischen der Controller noch Aufgaben wie Filterung und Sortierung vornehmen.



Bindings-Controller sind schizophren. Dadurch können sie zusätzliche Aufgaben wahrnehmen.

Diese Janusköpfigkeit ist wohl zunächst schwierig zu verstehen. Man kann sagen, dass die Daten über die Content-Bindings in den Controller gelangen und über die Eigenschaft `arrangedObjects` wieder (gefiltert usw.) an das View herausgegeben werden. Beachten Sie dabei bitte auch, dass die sortierten Daten des `arrangedObjects` ausgangs des Controllers stets als Array geliefert werden, da ja eine Reihenfolge besteht. Dies gilt auch dann, wenn eingangs des Arraycontrollers die Eigenschaft `contentSet` – also eine unsortierte Menge – gebunden wurde.

Einstellungen im Attributes-Inspector

Bei einem Arraycontroller gibt es eigentlich wenig zu erläutern.

Mode, Class Name, Entity Name

Zu beachten ist, dass er wie jeder Controller sozusagen in zwei Modi laufen kann, die man unter *Mode* in der Rubrik *Object Controller* des Attributes-Inspector einstellt:

- *Class*: Zum einen kann er Instanzen von (gewöhnlichen) Modelklassen verwalten. Es ist dann erforderlich, die Klasse anzugeben. Hierbei geht es weniger um die Bereitstellung von Daten als um die Actionmethoden zum Einfügen und Löschen von Instanzen. Diese müssen ja wissen, welche Klasse die erzeugte Instanz haben soll.
- *Entity Name*: Will man indessen Core-Data-Entitäten verwalten, so läuft er im Modus Entity. Er erzeugt dann nicht Instanzen von Klassen, sondern Instanzen von Core-Data-Entitätstypen. Dies sind natürlich auch Instanzen von Klassen, nämlich meist von `NSManagedObject`. Aber eine Entität kennt ihre Klasse, so dass die Angabe der Entität reicht. Dazu kommen wir im Abschnitt über Core Data.

In der Zeile darunter (*Class Name* bzw. *Entity Name*) ist dann der jeweilige Bezeichner einzutragen.

Übrigens merkt sich der Interface Builder, welche Schlüssel bereits verwendet wurden. Das erleichtert die Eingabe von Bindungen an ihn, da bereits Vorschläge für den Model-Key gemacht werden.

Gibt man als Klasse zudem `NSMutableDictionary` an, so werden automatisch Instanzen mit eben diesen Keys erstellt. Dies führt dazu, dass man Entitäten, deren Klasse man noch nicht programmiert hat, schnell simulieren kann (Rapid-Prototyping).

Always Use Multi Values Marker

Die Einstellung *Always Use Multi Values Marker* bedarf allerdings der Erläuterung. Wie Sie bereits bei den Bindings-Optionen gesehen hatten, können Placeholder-Texte gesetzt werden, wenn besondere Selektierungen vorliegen. Dies hatten wir ja auch für die leere Selektierung verwendet. Es gibt einen weiteren besonderen Fall, nämlich die Mehrfachselektierung. Wir benötigen sie nicht, da unser Tableview die Mehrfachselektierung nicht zulässt. Wenn dies allerdings so wäre, so ergäbe sich eine Fallunterscheidung:

- Sie haben eine Mehrfachauswahl, und die dargestellte Eigenschaft ist bei allen ausgewählten Einträgen gleich. Ein Beispiel sind etwa mehrere Umrechnungen, die alle denselben Umrechnungsfaktor haben.
- Sie haben eine Mehrfachauswahl, und die dargestellte Eigenschaft ist nicht bei allen ausgewählten Einträgen gleich. Es sind also mehrere Umrechnungen ausgewählt, die einen unterschiedlichen Umrechnungsfaktor haben.

Im zweiten Falle ist klar, dass eine echte Mehrfachauswahl vorliegt. Für eine Eigenschaft wird dann ein Multiselektions-Marker erzeugt, da sich ja die Eigenschaft nicht mehr einfach darstellen lässt. Diesem können wir eben in den Bindingsoptions einen Placeholder zuweisen.

Im ersten Falle ist es jedoch so, dass sich die Eigenschaft eigentlich noch darstellen ließe, eben durch den immer gleichen Wert. Dies macht der Arraycontroller auch standardmäßig. Wenn Sie die Option *Always Use Multi Values Marker* einschalten, wird dies nicht getan, sondern eben auch in diesem Falle der Marker erzeugt.

Observierbare Eigenschaften

An Eigenschaften, die für eine Bindung dienen können, haben Sie bereits `arrangedObjects`, `selection` und `canRemove` kennengelernt. Es gibt weitere:

Selektierungen

Die aktuelle Selektierung wird durch folgende Eigenschaften zur Bindung angeboten, die Sie selbstverständlich auch durch entsprechende Methoden im Code abfragen können:

- *selection* beschreibt die aktuelle Selektierung.
- *selectedObjects* ist ein Array mit den aktuell ausgewählten Instanzen. Es handelt sich auch dann um ein Array, wenn lediglich eine Instanz ausgewählt wurde.
- *selectionIndex* bezeichnet den Index des gewählten Eintrages.
- *selectionIndexes* gibt die Indexe (ja, im technischen Bereich verwendet man diese Pluralbildung!) einer Mehrfachselektion wieder.

GRUNDLAGEN

Hierbei wird als Collection ein sogenanntes Index-Set verwendet. Dies ist eine Ansammlung von Ganzzahlen. Der Vorteil gegen über einem Set liegt darin, dass man zum einen gleich Zahlen vom Typen `NSUInteger` erhält und nicht Number-Instanzen. Außerdem werden diese platzsparend gespeichert.

Wenn Sie aus dem Code heraus diese Eigenschaften abfragen, kann es zu Überraschungen kommen. Die von dem Arraycontroller gelieferten Instanzen können nämlich eine andere Klasse haben. Es handelt sich um sogenannte Bindings-Proxys. Auf der Webseite gibt es einen Artikel dazu. Aber schon hier: Auf jeden Fall können Sie die KVC-Methoden `–valueForKey:` usw. auf dieses Objekt anwenden.

Arranged-Objects

Bezogen auf den Inhalt gibt es zwei wichtige Eigenschaften, die als Angelpunkt für Bindings dienen können:

arrangedObjects sollten Sie zwischenzeitlich zur Genüge kennen. Diese arranged Objects existieren auch, wenn, wie bei uns, der Arraycontroller an ein Set gebunden ist. Die Reihenfolge ist dann freilich zufällig. Eine Sortierung kann aber sowohl über Sort-Descriptors im Arraycontroller erfolgen als auch über den Tableview.

Sort-Descriptors

Mit der Eigenschaft `sortDescriptors` wird ein Array von Instanzen der Klasse `NSSortDescriptor` an den Controller übergeben. Sie werden zuweilen im `–awakeFromNib` über ein

Outlet auf den Arraycontroller gesetzt. Ich gebe aber zu bedenken: Wenn die Einträge wie bei uns keine natürliche Reihenfolge besitzen, gibt es wenig Gründe, diese bereits im Arraycontroller anstelle des Tableviews sortieren zu lassen. Wenn es indessen bereits eine natürliche Reihenfolge gibt, so haben wir den Controller an ein Array gebunden, das bereits eine Sortierung mitbringt. Gleich folgt ein Beispiel zur alphabetischen Sortierung der Umrechnungsfaktoren.

Fangen wir mit der Implementierung der Sorterreihenfolge an: Zunächst bauen wir uns ein Outlet auf den Arraycontroller in der Klasse `Document`. Öffnen Sie Document.h:

```
@interface Document : NSPersistentDocument
@property (strong, nonatomic) IBOutlet NSArrayController* conversionsController;
@end
```

In der Implementierung müssen wir dann freilich etwas damit anfangen. Als richtigen Ort für die Initialisierung könnte man an `–awakeFromNib` denken. Allerdings haben Dokumente eine spezielle Stelle, nämlich `–windowControllerDidLoadNib:`

```
– (void)windowControllerDidLoadNib:(NSWindowController*)ctr
{
   [super windowControllerDidLoadNib:ctr];
   NSSortDescriptor* descriptor = [[NSSortDescriptor alloc] initWithKey:@"name"
                                                              ascending:YES];
   NSArray* descriptors = [NSArray arrayWithObject:descriptor];
   [self.conversionsController setSortDescriptors:descriptors];
}
```

Das ist eigentlich selbsterklärend: Man beachte lediglich, dass mehrere Sort-Deskriptoren angegeben werden können, die dann in der entsprechenden Reihenfolge abgearbeitet werden.

Jetzt müssen Sie das Outlet in Document.xib noch setzen, indem Sie eine Verbindung vom *File's Owner* (das ist ja hier das Dokument) zum Arraycontroller *Conversions Controller* ziehen. Außerdem muss im Attributes-Inspector des Arraycontrollers die Eigenschaft *Auto Rearrange Content* gesetzt werden, damit dieser automatisch bei einer Änderung die Reihenfolge ändert. Sie können die neue Funktionalität testen: Die einzelnen Einträge sollten sich jetzt automatisch sortieren.

HILFE

Sie können das Projekt in diesem Zustand als Projekt Converter-30 von der Webseite herunterladen.

Filter-Predicate

filterPredicate erlaubt uns die Filterung der angezeigten Elemente. Dies kann man zu einer Live-Suche verwenden. Auch dies implementieren wir gleich.

GRUNDLAGEN

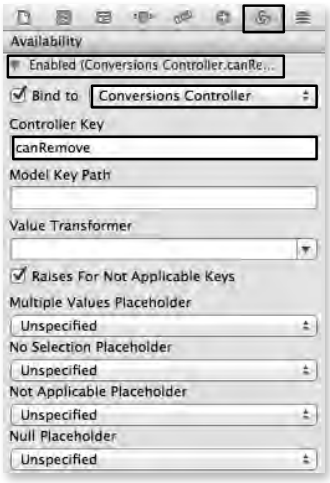
Prädikate sind Aussagen über Gegenstände, die je nach Gegenstand wahr oder falsch sein können, wie etwa »Der Umrechnungsfaktor ist kleiner als 1« oder »Die Bezeichnung beginnt mit 'Zo'«. Es werden also in guter alter Key-Value-Manier Eigenschaften (Umrechnungsfaktor, Bezeichnung) mithilfe von Operatoren (ist kleiner, beginnt mit) mit einem Wert (1, 'Zo') verglichen und geschaut, ob die Aussage zutrifft. Nimmt man aus einer bestehenden Menge nur diejenigen Elemente, für die die Aussage zutrifft, erhält man einen Filter.

Mit dem Filter-Predicate sorgen wir dafür, dass der Benutzer in unserem Programm suchen kann. Wie dies bei OS X üblich ist, machen wir das jedoch nicht durch einen Suchdialog, sondern durch eine Live-Suche. Das Suchfeld hatten wir ja bereits der Toolbar hinzugefügt.

Denken wir das mal durch: Der Arraycontroller bietet eine Eigenschaft `filterPredicate`, welche die Einträge filtert. Diese Eigenschaft lässt sich zur Bindung eines anderen Elementes verwenden, in diesem Falle zu der des Suchfeldes. Also müssen wir das Suchfeld in der Toolbar an diese Eigenschaft binden, damit es sich synchron zum Arraycontroller verhält. Gesagt, getan:

Öffnen Sie wiederum Document.xib und dort das Dokumentenfenster. Wir müssen das Suchfeld editieren, weshalb Sie es bitte in der Objektliste beim Fenster unter *Toolbar* | *Toolbar Item – Faktor suchen* | *Search Field* auswählen. (Das Öffnen der Toolbar im Fenster führt regelmäßig dazu, dass etwas völlig Kaputtes erscheint. Daher nehmen wir die Selektierung über die Objektliste vor. Eine Alternative wäre es, die Toolbar zunächst auf die oberste Ebene der Objektliste zu ziehen und nach getaner Arbeit wieder in das Fenster zu legen.)

Im Bindings-Inspector müssen Sie jetzt das *Predicate*-Binding wie in der Abbildung setzen:



Der Wert in einem Searchfield kann als Filterprädikat verwendet werden.

Das bedarf natürlich ein wenig der Erläuterung:

- Der *Display Name* dient der Bezeichnung der Art der Suche. Dies ist dann wichtig, wenn es mehrere Möglichkeiten zur Suche gibt, etwa nach Namen und nach Faktoren. Wir wollen hier ausschließlich nach der Eigenschaft `name` unserer Conversion-Instanzen suchen.
- Das *Predicate Format* bestimmt die Suche. Mit dem abgebildeten String geben wir an, dass wir nur diejenigen Einträge anzeigen wollen, die in der Eigenschaft *name* den vom Benutzer eingegebenen Text (*value*) enthalten (*contains*). Das *[cd]* sorgt dafür, dass die Suche case-insensitiv erfolgt, also ohne Berücksichtigung der Groß- und Kleinschreibung und diakritischer Zeichen (Umlaute und dergleichen).

Sobald Sie ein solches Binding gesetzt haben, erscheint ein neues Binding *Predicate 2*. Sie können hier eine andere Suche mit einem anderen Namen und einem anderen Prädikat eingeben. Der Benutzer kann dann in einem Menü auswählen, wonach er suchen will. Sie kennen dies sicher aus anderen Applikationen. Wir haben dafür keine Verwendung.

Wie Prädikate im Einzelnen funktionieren, wird von mir bei Core Data besprochen. Hier geht es mir nur um die Hausmacherart und die Binding-Möglichkeit des Arraycontrollers.

HILFE

Sie können das Projekt in diesem Zustand als Projekt Converter-31 von der Webseite herunterladen.

6.1.7 Der Tree-Controller

Ein Tree-Controller ist sozusagen das Bindingsgegenstück zum Outlineview. So wie dieses Hierarchien anzeigen kann, verwaltet der Tree-Controller sie.

Prinzipiell funktioniert ein Tree-Controller nicht anders als ein Arraycontroller. Er bietet jedoch kein Filter-Predicate, was Sie sicherlich schon in Applikationen bemerkt haben. Das macht ihn mir ein wenig madig.

Dafür enthält er zusätzliche Einstellungen für die Hierarchie in seinem Attributes-Inspector. Hier die wichtigsten:

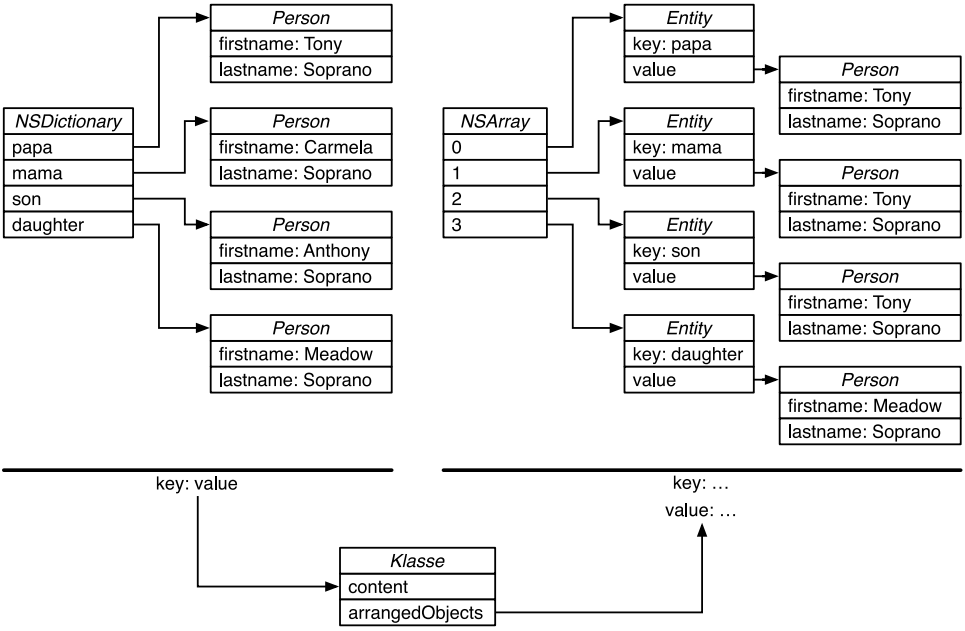
- `children` und die dazugehörigen Methoden `childrenKeyPath` und `setChildrenKeyPath:` bilden den Schlüsselpfad ab, der für die Einträge auf oberster Ebene die Kinderobjekte liefert. Die Model-Instanz muss entsprechend ein Array liefern, wenn diese Eigenschaft abgefragt wird.
- Die optionale Eigenschaft `count` (`countKeyPath`, `setCountKeyPath:`) kann gesetzt werden, um eine eigene Eigenschaft für die Berechnung der Anzahl der Einträge anzugeben. Bleibt das Feld leer, so holt sich der Tree-Controller die Kinderarrays

mittels der Eigenschaft `children` und und sendet dahin die Nachricht `count` (`NSArray`). Das Setzen dieses Schlüsselpfades bietet sich also an, wenn die Beschaffung des Arrays langsam ist, die Ermittlung der Anzahl indessen schnell.

- Die ebenfalls optionale Eigenschaft `leaf` und die entsprechenden Accessor-Methoden geben einen Schlüsselpfad an, unter dem der Tree-Controller erfahren kann, ob das entsprechende Element überhaupt Kinder hat. Auch hier kann dies nützlich sein, wenn die Beschaffung der Kinder teuer (langsam) ist, diese aber ohnehin nicht angezeigt werden sollen. Auch ist zu bedenken, dass Eltern und Kinder nicht der gleichen Klasse angehören müssen. In diesem Falle ist es ein einfacher Trick, beiden Klassen eine Eigenschaft zu geben, die für die Eltern `YES` antwortet, um das Aufklappen des Outlineviews zu ermöglichen, und für die Kinder `NO`, um dies zu verhindern.

6.1.8 Der Dictionary-Controller

Seit Leopard existiert dieser neue Controller als Subklasse von `NSArrayController`, der seine Daten nicht aus einem Array oder einem Set beschafft, wie es der Arraycontroller macht. Vielmehr ist die Datenquelle ein Dictionary. Auf der Seite zu einem Tableview hin existiert dann wieder die Eigenschaft `arrangedObjects`. Dabei macht der Dictionary-Controller aus einem Dictionary mit N Key-Value-Einträgen ein Array mit N Einträgen, wobei jeder Eintrag eine Entitätmit den Schlüsseln key und value ist. Die Klasse dieser Einträge im Array ist `NSObject` mit dem informellen Protokoll `NSDictionaryControllerKeyValuePair`.



Dictionary-Einträge werden gleichermaßen um 90 Grad gedreht.

In der obigen Abbildung würde man also etwa die Person für den Key @"papa" mit folgendem Code erhalten:

```
Person *papa = [family valueForKey:@"papa"];
```

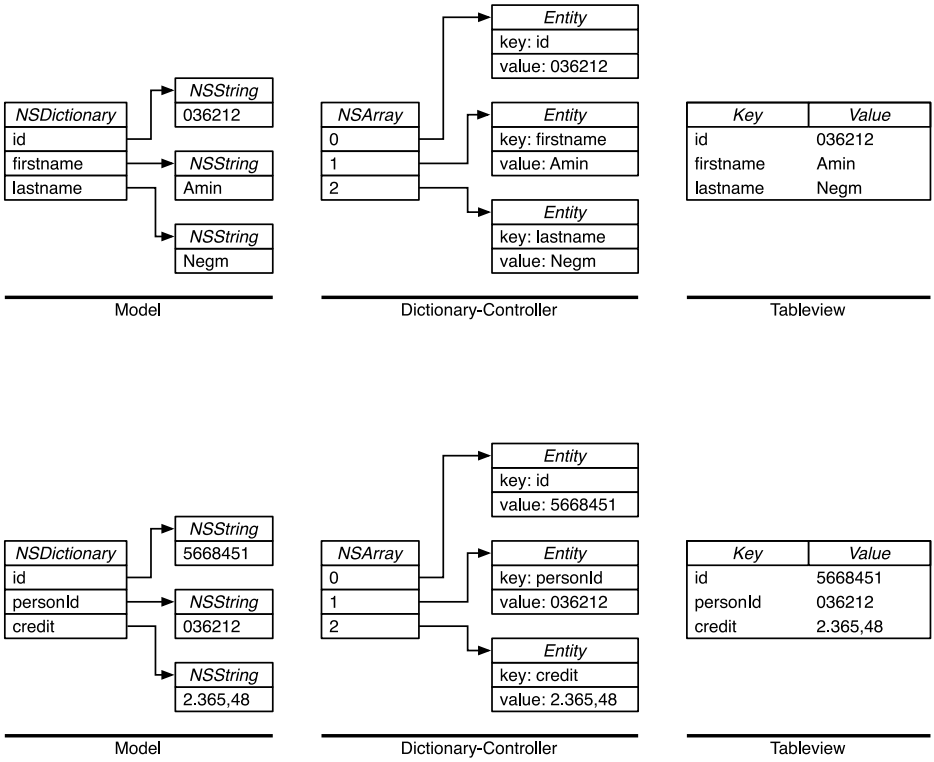
In den Arranged-Objects ist dies nur ein Eintrag, nehmen wir an der erste:

```
NSString *key = [[arrangedObjects objectAtIndex:0] valueForKey:@"key"];
Person *papa = [[arrangedObject objectAtIndex:0] valueForKey:@"value"];
```

Das sieht auf den ersten Blick kurios und kompliziert aus. Wozu braucht man so etwas? Zunächst einmal sehr selten, was auch erklärt, dass dieser Controller erst mit Mac OS X 10.5 geliefert wurde und damit der jüngste ist. Seine Besonderheit liegt in etwas anderem: Die anderen Bindings-Controller verwalten eine Mehrzahl von Entitäten, die allesamt über dieselben Schlüssel ansprechbar sind. Das ist etwa das Array von Personen. Hier wird nur ein Dictionary, also eine Entität, verwaltet, welches unbekannte Schlüssel beinhalten kann, sogar die Anzahl der Einträge kann unbekannt sein. Daher lassen sich nicht einzelne Tabellenspalten (wie viele?) an einzelne Schlüssel (welche?) binden. Vielmehr werden der Schlüssel und der dazugehörige Wert gebunden. Das funktioniert bei bei ganz unterschiedlichen Dictionarys.

Damit wären wir auch bei der Anwendung: Wenn Sie ein Dictionary erhalten, bei dem Sie nicht wissen, welche Schlüssel enthalten sind, hilft ein Dictionary-Controller. Nehmen Sie etwa an, dass Sie über eine Internetverbindung Daten erhalten, die mal eine Person in einem Dictionary mit den Schlüsseln `@"id"`, `@"firstname"` und `@"lastname"` enthalten und mal eine Kontobeschreibung, die die Schlüssel `@"id"`, `@"personId"` und `@"credit"` enthält. (Lesen Sie an dieser Stelle mal nach, was etwa ein REST-Service ist.) Sie könnten das nicht binden, weil für die Person drei Spalten mit den entsprechenden Schlüsseln gebunden werden müssten und bei einer Kontobeschreibung die Bindung auf andere Schlüssel lauten würde.

Stattdessen wird bei einem Dictionary-Controller ein Tableview mit zwei Spalten verwendet, welches an die Schlüssel `key` bzw. `value` gebunden wird. Hiermit lässt sich dann ein Dictionary mit unbekannten Schlüsseln darstellen.



Unterschiedliche Schlüssel in einem Tableview als Liste mit ihren Werten.

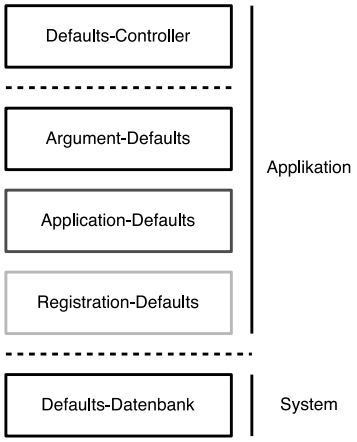
Beachten Sie bitte hierbei, dass die Schlüssel in einem Dictionary keine Sortierung aufweisen. Die Reihenfolge in dem Tableview ist daher zufällig. Man kann freilich auch hier eine Sortierung mittels Sort-Descriptoren angeben.

6.1.9 Der Defaults-Controller und Voreinstellungen

Eine besondere Form des Bindings-Controllers ist der Defaults-Controller. Er dient dazu, die Voreinstellungen (Defaults, Preferences) einer Anwendung zu verwalten. Dabei stellt er aber nur die halbe Wahrheit dar: Zusätzlich müssen wir nämlich noch im Code für die richtige Hintergrundmusik sorgen. Um das nicht zu zerfleddern, bespreche ich alles an dieser einen Stelle.

Defaultssystem

Defaults sind letztlich Property-Lists. In ihnen werden also mit dem Standardcontainer Informationen hinterlegt. Dabei besteht auf oberster Ebene ein Dictionary mit den verschiedenen Einstellungsgruppen. Natürlich kann man dann etwa auch Arrays speichern, was wir auch machen werden.



Die verschiedenen Defaults sind mehr oder weniger transparent in unserem Programm sichtbar. Dahinter werkelt ein Defaultssystem.

Grundsätzlich werden die Defaults vom System behandelt. Dabei kann es je nach Landeseinstellungen, Netzwerkumgebung usw. zu verschiedenen Defaultssätzen kommen. Dies ist aber Systemangelegenheit und für uns transparent.

GRUNDLAGEN

Mit transparent bezeichnet man eine Eigenschaft, Fähigkeit usw., die für den Programmierer nicht sichtbar ist. So liegt es hier: Der Programmierer bekommt seine Defaults. Dahinter versteckt sich ein kompliziertes System von OS X, welches den Programmierer jedoch nicht zu interessieren braucht, da es für ihn unsichtbar, eben transparent ist.

Nur fast gänzlich transparent sind drei sogenannte Default-Domains, die gestapelt sind. Grundsätzlich können zu einer Applikation vier Sätze von Defaults relevant sein:

- *Registration-Domain* enthält üblicherweise feste Werkseinstellungen. Diese gelten also, wenn der Anwender keine Änderungen vorgenommen hat.
- Eine *Application-Domain* enthält Änderungen des Benutzers für die Anwendung. Diese gehen der Registration-Domain vor.
- Zudem können bei Programmstart Parameter übergeben werden, die nur für den einzelnen Programmlauf Defaults noch mal überschreiben: *Argument-Domain*.
- Der *Defaults-Controller* ist nicht für unsere Arbeit im Programm wichtig, sondern für die bindingskompatible Anbindung von Views, insbesondere im Einstellungsfenster.

POWER

Darüber hinaus existiert seit OS X 10.7 die Möglichkeit, Schlüssel-Werte-Paare in der iCloud zu speichern und so auf verschiedenen Geräten eines Nutzers zur Verfügung zu stellen. Hierfür existiert die Klasse `NSUbiquitousKeyValueStore`. Programmierung mit der iCloud ist der 2. Auflage des zweiten Bandes vorbehalten.

Wenn wir also in unserem Programm einzelne Einstellungen abfragen, fallen wir nacheinander durch die Domänen, bis ein Treffer gefunden wird. Wir erfahren aber grundsätzlich nicht, aus welcher Domain der Treffer stammt. Hierbei ist es zudem noch wichtig, dass das Defaultssystem nicht ständig geänderte Einstellungen auf die Platte zurück schreibt. Vielmehr geschieht dies unregelmäßig, im Bedarfsfalle spätestens bei Beendigung der Anwendung. Sie sollten also in diesem Kapitel die Anwendung stets ordnungsgemäß über das Applikationsmenü *CocoaConverter* mit *Quit New Application* beenden!

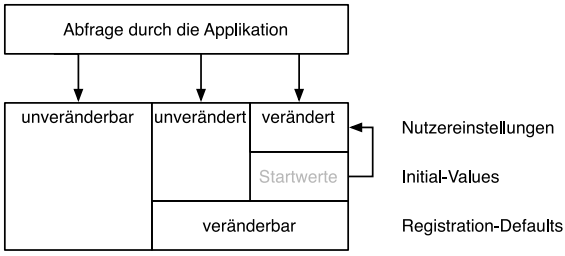
GRUNDLAGEN

Darüber hinaus gibt es Defaults, die sich nicht auf unsere Anwendung beziehen, sondern global wirken oder etwa die Landeseinstellungen betreffen. Diese können gesondert abgefragt werden, etwa mit der bereits bekannten Klasse `NSLocale`. Wir konzentrieren uns hier auf die oben genannten Domains.

Registrationdefaults und Application-Delegate

Umgekehrt erfahren wir von der Existenz der verschiedenen Domains, wenn wir die Registration-Defaults setzen. Es ist nämlich erforderlich, dass wir für jede Einstellung in unserem Programm zunächst einen Standardwert angeben. Dies muss vor deren Benutzung erfolgen. Wir machen das auch gleich mit der Methode `–registerDefaults:` (`NSUserDefaults`).

Eine weitere Stufe stellen die sogenannten Initial-Values dar, die man mit der Methode `–setInitialValues:` (`NSUserDefaultsController`) setzen kann. Hierbei handelt es sich um die Standardeinstellungen für diejenigen Defaults, die der Benutzer verändern kann. Es ist nämlich ebenfalls sinnvoll, sich selbst Defaults anzulegen, die der Benutzer nicht ändern darf. Denken Sie etwa an den Fall, dass Sie mit einem bestimmen Webserver kommunizieren wollen und die URL speichern müssen. Einfach in den Defaults ablegen. Dem Benutzer geben wir aber kein User-Interface zur Änderung. Also: Diese Defaults müssen in den Registration gespeichert sein, tauchen in den Initial-Values jedoch nicht auf. Für eine Abfrage der Defaults aus unserem Programm heraus spielen sie indessen keine Rolle.



Alle Defaults benötigen eine Registrierung. Zu den editierbaren können wir Standards vorgeben.

Die Angabe der Initial-Values ist nicht zwingend erforderlich. Wenn man sie angibt, so kann man mit der `Methode –revertToInitialValues (NSUserDefaultsController)` schnell zu diesen zurückkehren. Sie kennen das aus Einstellungsfenstern, in denen ein Button *Zurücksetzen* oder ähnlich existiert.

Ein möglicher Standardort für die Registrierung ist das Application-Delegate, mit dem wir uns später noch gesondert beschäftigen werden. Werden Defaults nur in Teilen der Anwendung benutzt, so bietet es sich an, einen entsprechenden Controller oder sogar die nutzende Klasse als Standort für die Registrierung auszuwählen. Das System ändert sich dadurch nicht. An dieser Stelle erzeugen Sie sich bitte einfach eine entsprechende Subklasse von *NSObject<NSApplicationDelegate>*, die Sie *AppDelegate* nennen, in Xcode über *File* | *New* | *File…* Als Vorlage wählen Sie bitte *Objective-C class*.

AUFGEPASST

Gab es die Klasse und eine dazugehörige Instanz nicht schon? Ja, das war aber ein anderes Projekt, nämlich eines ohne Dokumente. Dort befindet sich bereits ein Application-Delegate in der Projektvorlage. Jetzt haben wir es aber mit einem Projekt zu tun, welches Dokumente unterstützt. Da müssen wir das selbst erzeugen.

Wir machen darin gar nichts Berühmtes, sondern implementieren nur eine einzige Methode:

```
…
@implementation AppDelegate
+ (void)initialize
{
   if( self != [AppDelegate class] ) {
      return;
   }

   NSDictionary *registerDefaults
   = [NSDictionary dictionaryWithObjectsAndKeys:@"Amin", @"author", nil];

   NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
   [defaults registerDefaults:registerDefaults];
}
@end
```

GRUNDLAGEN

Die Klassenmethode `+intialize` ist bereits in Kapitel 4 besprochen worden. Schauen Sie dort notfalls noch einmal nach.

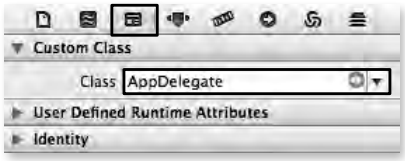
Um die Defaults zu registrieren, erstellt die Methode also ein Dictionary, welches für den Schlüssel author einen String Amin speichert. Jeder Defaultwert wird also unter

einem Schlüssel gespeichert. Es handelt sich ja um eine Property-List. Dabei muss diese Liste nicht vollständig sein. So können etwa zusätzlich dokumentenbezogene Defaults in `+initialize` (Document) gesetzt werden. Wichtig ist aber, dass die Registrierungsdefaults vor ihrer Benutzung gesetzt sind.

TIPP

Natürlich kann man daran denken, die Registerdefaults einfach als Property-Lists zu speichern und entsprechend zu laden. Und ja, das ist zuweilen auch sinnvoll, jedenfalls modulweise. Wir müssen dazu aber Property-Lists dem Projekt hinzufügen. Im Kapitel über Xcode werden Sie das genauer lernen. Ich will hier aber nicht von Höckchen auf Stöckchen springen.

Öffnen Sie jetzt *MainMenu.xib* (nicht: *Document.xib*). Zunächst müssen wir unser Application-Delegate im Nib instantiieren. Hierzu ziehen Sie aus der Library nach einer Suche mit *Object* den blauen Würfel aus der Library in die Objektliste. Im Identity-Inspector muss dann noch unter *Custom Class* | *Class* die Klasse *AppDelegate* eingetragen werden. Weil man es so selten macht, hier noch einmal ein Screenshot:



Dieser Custom-Controller bedient unser Application-Object.

Jetzt müssen wir freilich noch mitteilen, dass diese Instanz unser Application-Delegate sein soll. Dazu setzen wir ein Outlet vom File's Owner, der in MainMenu.xib die Application ist, zu unserem Delegate. (Sie wissen schon, mit gedrückter [ctrl]-Taste eine Verbindung ziehen. So langsam sollte ich das nicht mehr erwähnen müssen.)

Rekapitulieren wir noch einmal kurz, was geschieht:

- Wir haben uns eine neue Klasse `AppDelegate` geschaffen.
- Diese wird im MainMenu.xib instantiiert.
- Daher wird zunächst die Methode `+initialize` der Klasse aufgerufen.
- Diese registriert einen Defaultwert.
- Die Verbindung von Application (File's Owner) zum Delegate ist hier eigentlich noch nicht erforderlich, weil alle vorgenannten Schritte auch ohne dies funktionieren. Aber wenn wir uns schon ein Delegate machen, sollten wir es auch setzen.

Ziehen Sie jetzt ein neues Fenster in die Objektliste, und geben Sie diesem in dem Attributes-Inspector unter dem Eintrag *Title* den Namen *Preferences*. Außerdem sorgen Sie dafür, dass im Attributes-Pane des Inspectors für dieses Fenster kein Haken vor *Visible At Launch* und *Release When Closed* gesetzt ist. Das Ausschalten der ersten Option sorgt dafür, dass dieses Fenster nicht automatisch geöffnet wird, die zweite, dass es mehrmals

nacheinander geöffnet werden kann. Dies entspricht ja auch dem erwarteten Verhalten eines Preferences-Fensters.

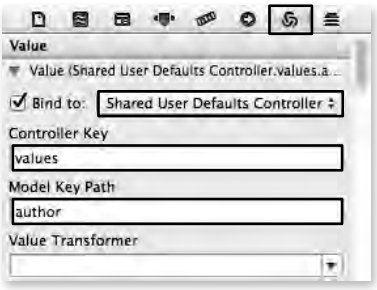
AUFGEPASST

Im Abschnitt über Windowcontroller lernen Sie allerdings eine elegantere Art kennen, die modularer ist und den Speicher schont.

Ziehen Sie ein Label und ein editierbares Textfeld in das Fenster. Das Label benennen Sie mit *Ersteller:*.

Gut, wir sind im Kapitel über Bindings. Und jetzt kommen wir zu dieser Hälfte. Wechseln Sie für das Textfeld in den Bindings-Inspector.

Setzen Sie jetzt das Value-Binding auf den bereits vorgewählten Eintrag *Shared User Defaults Controller,* und binden Sie die Eigenschaft *values* mit dem *Controller Key values* und dem *Model Key Path author.* Auch hier noch einmal das Ergebnis:



Wir binden das Textfeld an einen ganz besonderen Controller für User-Defaults.

Sie werden bemerken, dass in der Objektliste automatisch ein neues Objekt *Shared User Defaults Controller* aufgetaucht ist. Wie Sie an dem shared erkennen können, wird dieses geteilt; geteilt im Sinne von geteiltem Leid, nicht im Sinne von dividiert. Es handelt sich um einen Singleton, den es nur einmal gibt.

GRUNDLAGEN

Auch in dem vorher abgebildeten Code lag ein solcher Singleton vor, den wir uns mit `–standardUserDefaults` besorgten. Die Einstellungen werden also standardmäßig über einen einmaligen zentralen Defaultstack verwaltet, wie Sie ihn in der Graphik eingangs des Abschnittes sehen konnten.

Auch hierbei handelt es sich daher nicht um eine Instanz im Nib im eigentlichen Sinne, sondern um einen Verweis auf einen Singleton, auch wenn das aus technischen Gründen anders einsortiert ist. Tatsächlich gibt es diesen allerdings nur einmal für alle Nibs und zwei gleichgültig, ob ein Nib mehrfach geladen wird oder aber er in verschiedenen Nibs auftaucht.

GRUNDLAGEN

Der Hintergrund ist in etwa, dass Singletons auf die übliche Art und Weise instantiert werden können, sie jedoch dann immer eine Referenz auf ein und dieselbe Instanz liefern. Mehrere Nibs benutzen dann also jede ihre eigene Instanz, die eben bloß für alle dieselbe ist. Ja, ja, ich weiß, recht wirr. Und man kann sich auch über den Sinn und Unsinn von Singletons streiten.

Da sich das Fenster nicht automatisch öffnet, müssen wir es explizit dazu bringen. Dazu öffnen Sie – wenn nicht bereits geöffnet – in der Objektliste mit einem Klick die Menüleiste. Klicken Sie dort auf *Converter*, so dass sich das Menü öffnet. Nun ziehen Sie wiederum eine Action-Verbindung von dem Menüeintrag *Preferences* zu dem Fenster *Preferences* in der Objektliste. Loslassen und im aufspringenden HUD die Methode `–makeKeyAndOrderFront:` anwählen. Diese sorgt dafür, dass ein noch nicht geöffnetes Fenster geöffnet, das Fenster nach vorne geholt wird und das Fenster den Fokus für Tastatureingaben erhält. Umgangssprachlich: Das Fenster wird im Vordergrund geöffnet.

Starten Sie die Applikation. Wenn Sie jetzt die Voreinstellungen unseres Programmes über das Menü öffnen, sehen Sie den voreingestellten String im Fenster. Beenden Sie das Programm mit *Quit*. (Nicht in Xcode mittels des *Stop*-Items in der Toolbar abschießen!)

Wir haben also gesehen, dass der registrierte Defaultwert unter seinem Schlüssel author im Preferencesfenster dargestellt werden kann. Um weitere Forschungen zu betreiben, starten Sie bitte das Programm *Terminal*, welches sich auf der Festplatte im Ordner *Programme/Dienstprogramme* befindet – oder Sie ganz einfach mit Spotlight suchen. Hierbei handelt es sich um ein kleines Programm, mit dem Sie Kommandozeilenprogramme starten können. Das machen wir jetzt auch. Geben Sie

```
$defaults read com.cocoading.Converter[Enter]
```

ohne das *$* ein

HILFE

Die Ausgabe links von der Schreibmarke nennt man Prompt. Sie zeigt an, dass man einen Befehl eingeben kann. Allerdings kann diese konfiguriert werden, weshalb ich nicht genau weiß, was bei Ihnen an dieser Stelle steht. Standard ist *Rechnername*:*Verzeichnis Nutzername*. Üblicherweise kürzt man das mit einem $ ab, wie ich es hier getan habe. Deshalb sollen Sie das auch nicht eingeben.

Das Kommandozeilenprogramm defaults erlaubt es uns, die Einstellungen von Programmen anzuzeigen und zu verändern. In diesem Falle geben wir mit dem Befehl *read* an, dass die Einstellungen gelesen werden sollen. Als Parameter bekommt es zudem den Namen unserer Applikationsdomain com.*yourcompany*.CocoaConverter. Wie wir diesen ändern, erfahren Sie wiederum im Kapitel über Xcode.

HILFE

Sie haben möglicherweise die Applikation anders benannt. Um das überprüfen zu können, selektieren Sie bitte in der Projektleiste den Projekteintrag ganz oben und wählen dann rechts daneben in der Liste den Eintrag unterhalb von *Targets* aus. Rechts wählen Sie bitte Summary aus. In der zweiten Zeile finden Sie den Eintrag *Bundle Identifier*. Dieser ist maßgeblich und muss von Ihnen im Terminal hinter dem *read* angegeben werden.

Es erscheint eine Einstellung, die allerdings gerade unser `author` nicht enthält:

```
{
    NSNavLastRootDirectory = "~/Desktop";
}
```

Hier wurden bereits von Cocoa selbst eine Einstellungen hinterlegt, nämlich zum letzten gewählten Speicherort. (Cocoa-Applikationen erinnern sich ja daran.)

AUFGEPASST

Es kann auch sein, dass nur eine Fehlermeldung … *does not exist* erscheint. Das liegt dann daran, dass Sie noch nie ein Dokument gespeichert haben und daher bisher überhaupt kein Eintrag für unsere Applikation in der Defaults-Datenbank erstellt wurde. Das ist gleichgültig, da uns der existierende Eintrag von Cocoa ohnehin nicht interessiert. Gehen Sie wie beschrieben weiter vor.

Dass unser Schlüssel author nicht angezeigt wird, liegt daran, dass wir bisher nur Registration-Defaults haben. Diese befinden sich aber kodiert in unserem Programm, so dass sie nicht gespeichert werden müssen. Deshalb kennt defaults diese nicht.

Wir müssen also mal einen Wert in den Defaults ändern. Dazu starten Sie wiederum unsere Applikation (das Terminal lassen Sie bitte offen) und geben in den Einstellungen anstelle von *Amin* einfach *Negm-Awad* ein. Wieder das Programm ordnungsgemäß mit *Quit* beenden. Wenn Sie jetzt den Befehl eingeben, wird die neue Einstellung erkannt:

```
$defaults read com.cocoading.Converter
{
    NSNavLastRootDirectory = "~/Documents";
    author = "Negm-Awad";
}
```

Wunderbar! Jetzt starten Sie wieder unsere Applikation und können erkennen, dass sich dieser Wert tatsächlich auch dort wieder findet. Noch einmal beenden.

Jetzt löschen wir im Terminal einmal die gespeicherten Defaults:

```
$defaults delete com.cocoading.Converter
$defaults read com.cocoading.Convert
… Domain com.cocoading.Converter does not exist
```

Aha, sie sind also verschwunden. Wieder das Programm starten. Siehe da: Auch im Preferencesfenster taucht wieder der alte Wert *Amin* auf.

Erkenntisse:

- Wir können in unserer Applikation Defaultwerte registrieren. Diese werden dann zur Laufzeit aus der Applikation erzeugt.
- Wenn der Benutzer Änderungen vornimmt, so werden diese in der Defaultsdatenbank gespeichert. Dies lässt sich mit dem Kommandozeilenprogramm defaults anzeigen und bearbeiten.
- Wenn sich dort ein abweichender Wert befindet, so wird dieser verwendet. Ist dies nicht der Fall, so gilt der Wert aus den Registrationdefaults.

HILFE

Sie können das Projekt in diesem Zustand als Projekt Converter-32 von der Webseite herunterladen.

6.1.10 Komplexe Bindings

Es gibt Bindings, die einen am Anfang herausfordern. Und dabei kann man in der Regel zwei Fälle unterscheiden: Die Hintereinanderschaltung von Bindings-Controllern und die Auswahl aus einer Liste als Wert eines anderen Objektes.

Bindingsketten

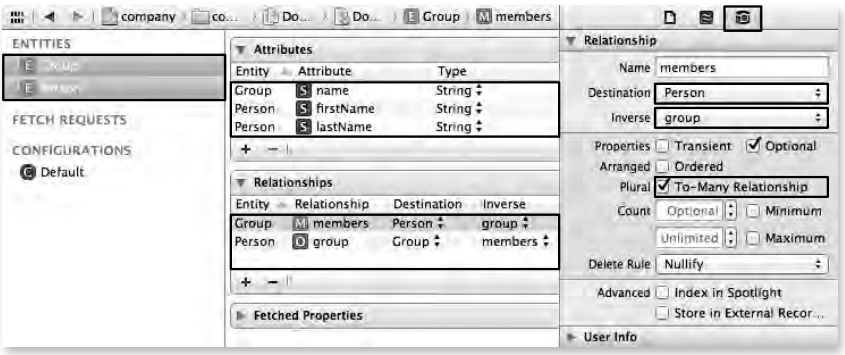
Denken wir uns eine einfache Datenstruktur aus, die Bindingsketten erfordert. Ach nein, brauchen wir uns gar nicht auszudenken, hatten wir nämlich schon: unser Firmenbeispiel aus dem Abschnitt zur Speicherverwaltung.

Kurz rekapitulieren: Wir hatten eine Firma, die verschiedene Gruppen kannte. Zu jeder Gruppe gehörten wieder mehrere Mitglieder. Schauen Sie sich vielleicht noch einmal die entsprechenden Graphiken an, und zwar sowohl als ERM als auch als lebende Instanzen.

Ach, bauen wir uns doch einmal schnell ein Core-Data-Projekt damit. Schließen Sie zunächst das offene Projekt in Xcode. Erzeugen Sie dann in Xcode mit *File* | *New* | *Project…* ein neues Projekt. Wählen Sie *Cocoa Application* aus. Nach einem Klick auf *Next* geben Sie als *Product Name* einfach *Company* ein und schalten bitte den Support für Core Data

und Dokumente ein. Auch Automatic Reference Counting bleibt bitte eingeschaltet. Als *Document Extension* geben Sie bitte wenig überraschend *company* ein. *Next* und speichern.

Nach der Erzeugung öffnen Sie bitte durch Auswahl von *Company* | *Company* | *Document.xcdatamodel* im Project-Navigator das Modell und legen hierzu zwei Entitäten *Group* und *Person* an. Geben Sie wie unten abgebildet den Entitäten Attribute und legen außerdem bei jeder Entität eine Relationship zu der jeweils anderen an, ebenfalls wie unten abgebildet. Die Beziehung `members` von der Gruppe zur Person muss eine To-Many-Relationship sein. Denken Sie bitte daran, dass die Beziehung *members* von der Entität *Group* eine To-many-Beziehung ist und als Inverse die Eigenschaft *group* der Entität *Person* hat. Das Ergebnis sollte in etwa so aussehen:



Selektiert man zwei Einträge in der Entitätenliste, erhält man eine kombinierte Ansicht.

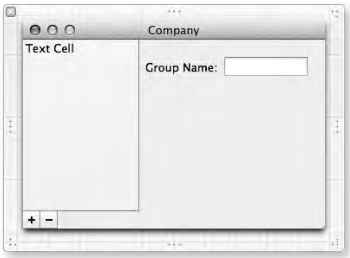
HILFE

Zur Erinnerung: In der linken Liste erscheinen oben die Entitäten. Sie können der Liste mit einem Klick auf + *Add Entity* Elemente hinzufügen. Rechts erscheinen bei ausgewählter Entität oben die Attribute und darunter die Beziehungen. Hier können Sie jeweils mit dem unter der Liste befindlichen + einen Eintrag hinzufügen. Haben Sie zwei Entitäten ausgewählt, erzeugt Xcode automatisch je Entität einen neuen Eintrag.

Sie sollten auch in jeder Entität mindestens ein Attribut mit einem Defaultwert besetzen, da sich sonst neue Einträge im User-Interface nur schwer erkennen lassen.

Bauen wir uns wieder ein simpelstes User-Interface dazu: Document.xib öffnen. Benennen Sie bitte das Fenster mit *Company* (sowohl als *Window* | *Title* im Attributes-Inspector wie auch als *Identity* | *Label* im Identity-Inspector). Im Fenster erzeugen Sie links ein einspaltiges Tableview mit zwei Buttons für Hinzufügen und Löschen. Rechts bringen Sie zunächst nur ein Textfeld unter, gegebenenfalls mit Label. Ich habe beim Tableview gleich wieder für die Sourcelist-Einstellungen (bei *Highlight* im Attributes-Inspector) gesorgt und das Autolayout aller Elemente konfiguriert. Wichtig ist dies freilich nicht für die Funktionalität, die

wir hier besprechen wollen. Aber: Übung macht den Meister! Außerdem können Sie wieder für das Tableview ganz unten im Attributes-Navigator den Fokusring ausschalten.



Das Grundfenster für die Gruppen des Unternehmens

Sie ahnen es schon: Ziehen Sie einen Arraycontroller in die Objektliste und benennen Sie ihn im Identity-Inspector mit *Groups Controller*. In seinem Attriutes-Inspector schalten wir ihn in den Mode *Entity* und geben als Bezeichnung der Entität *Group* ein. Außerdem setzen Sie das Häkchen vor *Prepares Content*.

Im Dokumente-Fenster verbinden Sie jeweils die beiden Buttons mit den Actionmethoden add: bzw. remove: des neuen Arraycontrollers.

Das Binding des Arraycontrollers schauen wir uns jetzt aber mal etwas genauer an: Wir binden lediglich den *managed Object Context*, indem Sie im Bindings-Inspectors dieses Binding öffnen und als Ziel den *File's Owner* eintragen sowie *managedObjectContext* als *Model Key Path*. Wie ich Ihnen schon in Kapitel 2 sagte, ist dies sozusagen eine große Halde von Core Data, in der eben Instanzen abgelegt werden. Wir binden aber keines der Content-Bindings! Daher weiß der Arraycontroller nicht, woher er die Instanzen beziehen soll und nimmt einfach alle, die sich zu seiner Entität im Kontext befinden. Dies ist auch richtig, da wir tatsächlich alle Gruppen sehen wollen. Ermöglicht wird das übrigens durch die Einstellung *Prepares Content*.

Selektieren Sie jetzt die Tabellenspalte (wirklich die Spalte, am besten in der Objektliste!) und binden Sie deren *Value* wie folgt:

```
Bind To: Groups Controller
Controller Key: arrangedObjects
Model Key Path: name
```

Den *Value* des Textfeldes binden Sie an die Selektion des Arraycontrollers:

```
Bind To: Groups Controller
Controller Key: selection
Model Key Path: name
```

Das Programm starten, um die bisherigen Arbeiten zu testen: Einfügen, löschen, ändern! Funktioniert alles? Gut!

HILFE

Ich möchte eigentlich, dass Sie sich mal selbst da durch beißen. Wenn Sie nicht weiterkommen, schauen Sie ruhig noch einmal in Kapitel 2 nach. Kommen Sie gar nicht zurecht, können Sie sich das Projekt in diesem Zustand als Company-01 von der Webseite herunterladen. Aber wirklich: Versuchen Sie sich zunächst selbst!

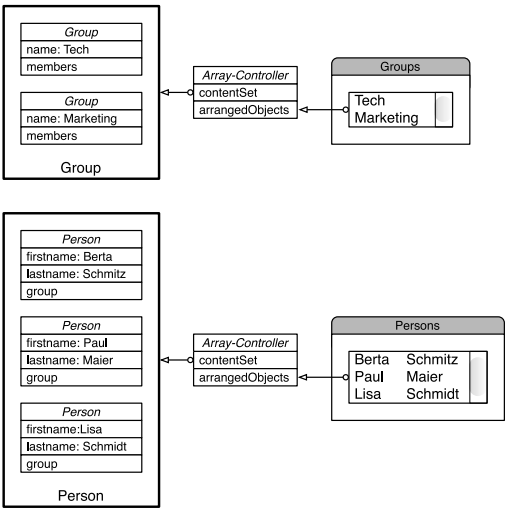
Ziehen Sie jetzt ein zweites Fenster in die Objektliste von Document.xib, und benennen Sie es *Persons*. Achten Sie im Attributes-Pane darauf, dass das Häkchen vor *Visible At Launch* aktiviert ist. Bauen Sie bitte ein Tableview mit zwei Spalten nebst Buttons herein. Bei Personen haben wir ja zwei Eigenschaften: `firstName` und `lastName`. Das Tableview habe ich hier nicht als Sourcelist konfiguriert.

Natürlich muss dafür auch ein Arraycontroller ins Hauptfenster, den Sie bitte *Persons Controller* nennen. Den stellen Sie ebenso ein wie den Groups-Controller, wobei Sie natürlich als Entität *Person* wählen. Denken Sie an das *Prepares Content* und das Binding für den *managed Object Context*, wobei Sie bitte darauf achten, dass bei *Binds To:* wirklich *File's Owner* eingestellt ist. (Der Default steht hier auf *Groups Controller*.)

Nun noch die Spalten des Tableviews an den neuen *Person Controller* binden und die Buttons mit den Actionmethoden des Controllers. Wieder starten und testen.

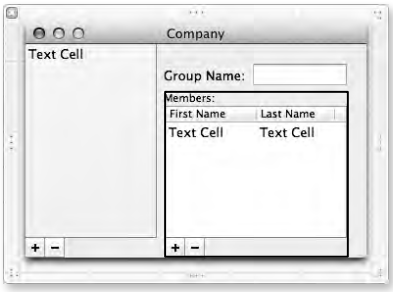
Dennoch: Das ist nicht das, was Sie erwartet haben. Sie können nämlich jetzt Abteilungen anlegen, Sie können auch Personen anlegen, Sie können das aber nicht in Beziehung zueinander setzen.

Dies ist auch klar, wenn man sich das Ganze mal aufzeichnet:



Ein Abbild der modernen westlichen Gesellschaft: jeder für sich und daher beziehungslos.

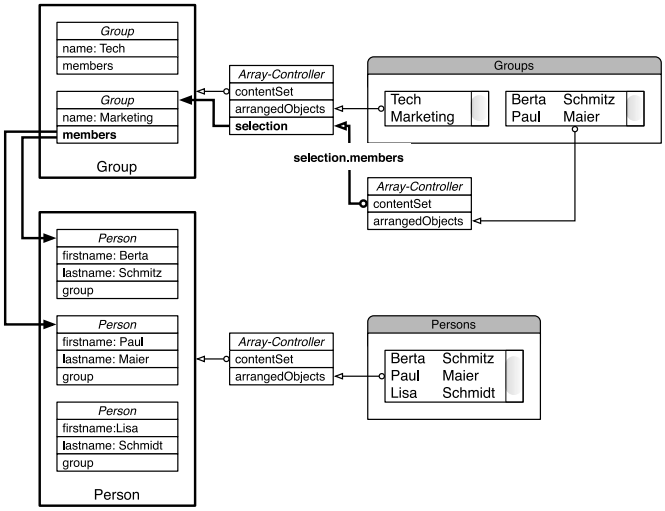
Okay, als Erstes benötigen wir dazu irgendwie ein User-Interface. Ziehen Sie bitte ein weiteres Tableview nebst Buttons in das Company-Fenster, und zwar unterhalb des Gruppennamens. Bauen Sie dieses (graphisch) genau auf wie die Personenliste im Persons-Fenster. Wegen des doch recht komplexen Aufbaus und um Missverständnisse zu vermeiden, hier noch einmal ein Screenshot:



Die Personen, die Mitglieder einer Gruppe sind

Jetzt können Sie natürlich das Tableview wieder so wie im Personsfenster binden. Das Resultat wäre aber dann ja dasselbe: Es würden alle Personen erscheinen. Daran ändert sich ja nichts, bloß weil wir das Fenster gewechselt haben.

Überlegen wir mal: Wir benötigen genau diejenigen Personen, die zu der jeweiligen Gruppe gehören. Dies ist die Eigenschaft members einer bestimmten Gruppe, nämlich der links im Sourceview selektierten. Eine bestimmte Gruppe erhalten wir über die observierbare Selection-Eigenschaft des Arraycontrollers für die Gruppe. Das hat beim Gruppennamen ja auch funktioniert.



Achten Sie auf die fetten Linien und Texte: Wir holen uns als Content nur diejenigen Personen, die sich in der Eigenschaft members der selektierten Gruppe befinden.

Jetzt ziehen Sie demnach bitte einen weiteren Arraycontroller in die Objektliste und bezeichnen diesen als *Members Controller*. Auch hier setzen Sie den Modus *Entity* und als Entität *Person*. Das ist so richtig, da dieser Controller weiterhin Personen verwalten soll, allerdings bestimmte Personen, nämlich die Mitglieder einer bestimmten Gruppe. Das Häkchen *Prepares Content* lassen Sie diesmal weg! Denn wir werden den Content binden. Damit bezieht er seinen Inhalt aus diesem Binding und hat selbst nichts vorzubereiten. Daher binden Sie bei diesem den *managed Object Context* des Members-Controllers in gewohnter Manier und zusätzlich eben das Binding *Content Set*:

```
Bind To: Groups Controller
Controller Key: selection
Model Key Path: members
```

Das Tableview im Company-Fenster für die Gruppenmitglieder bitte dann an diesen Controller anstelle des Persons-Controllers binden. Die Model-Keys `firstName` und `lastName` verwenden Sie jedoch bitte weiterhin. Außerdem verbinden Sie die Buttons darunter mit den entsprechenden Actionmethoden des neuen Members-Controllers.

Übersetzen, starten und testen! Legen Sie zwei Gruppen an und fügen Sie im Groups-Fenster diesen jeweils Personen hinzu, denen Sie zur besseren Unterscheidbarkeit irgendwelche Namen geben. Bei einem Wechsel in das Personenfenster bemerken Sie den Bezug. Sie sehen ebenfalls im Persons-Fenster, wie die in der Gruppe angelegten Personen dort auch erscheinen. Dieses zeigt eben weiterhin sämtliche Personen an, unabhängig von der Gruppe.

Aber, sehen Sie das kleine Problem? Wenn Sie wieder eine Person aus der Gruppe löschen, dann verschwindet sie zwar dort einwandfrei, bleibt aber im Persons-Fenster erhalten. Das hat einen einfachen Grund: Der gebundene Members-Controller kann ja zwei Möglichkeiten wahrnehmen:

- Die Person wird aus der Gruppe entfernt. Die Instanz bleibt dabei aber im Kontext erhalten.
- Die Person wird auch aus dem Kontext gelöscht.

Dabei kann man nicht davon sprechen, dass das eine richtig sei, das andere falsch. Vielmehr verhält es sich so, dass es auf die Anwendung ankommt. In unserem Beispiel liegt es nahe, die Personen insgesamt zu löschen. Aber wären die Gruppen so etwas wie Wiedergabelisten in iTunes, so wäre das sicher falsch: Bloß weil ich ein Lied aus der Wiedergabeliste lösche, soll es nicht insgesamt entfernt werden. Sie sehen also: Tatfrage!

Daher kann man das einstellen. Schauen Sie sich noch einmal bei selektiertem Members-Controller im Bindings-Inspector die Einstellungen für das Content-Set an: Dort gibt es die Option *Deletes Objetcs On Remove*. Wenn Sie diese anwählen, dann wird tatsächlich die Person nicht nur aus der Gruppe entfernt, sondern auch aus dem Kontext gelöscht. Probieren Sie es aus! (Und stellen Sie es danach wieder aus.)

Hier kurz wieder die Handwerksregeln:

- Ein Arraycontroller verwaltet eine Vielzahl von Instanzen.
- Wollen wir *sämtliche* Instanzen einer Entität verwalten lassen, so setzen wir die Option *Prepares Content* und lediglich das Binding *Managed Object Context*.
- Wollen wir indessen ganz bestimmte Instanzen verwalten lassen, die sich aus einer Beziehung in einem anderen Objekt ergeben, so setzen wir ein Content-Binding auf diese Beziehung. (Wird das Managed-Object-Context-Binding nicht gesetzt, so stiehlt es der Arraycontroller aus dem selektierten Objekt, was in der Regel gewollt ist.)
- Auf jeder Ebene einer To-many-Beziehung benötigen wir einen weiteren Arraycontroller. Würden wir unsere Mitglieder also wieder auf Einsatzorte verweisen, würden wir diese Eigenschaft wieder mittels eines Arraycontrollers verwalten, wobei das Content-Binding eben auf die gerade ausgewählte Person mit dem Model-Key-Path *Einsatzorte* lauten würde.

Verdeutlichen Sie sich vielleicht noch einmal die Unterschiede in der Konfiguration des Persons-Controllers einerseits und des Members-Controller andererseits.

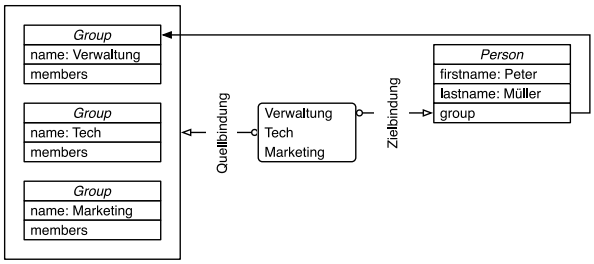
HILFE

Sie können das Projekt ist in diesem Zustand als Company-02 von der Webseite herunterladen.

Selektions-Bindings

Das Ganze sieht ja schon schön aus. Es gibt aber noch ein Binding, welches den Anfänger zur verzweifelten Weißglut bringen kann: das Auswahlbinding.

Die Problemstellung ist einfach: Ich habe eine Auswahlmenge und ein Zielobjekt. Und ich will jetzt eine Beziehung vom Zielobjekt auf ein (selektiertes) Objekt aus der Auswahlmenge zuweisen. Klingt nach einem Pop-up, nicht wahr? Wenn man dies allerdings binden will, ist Schluss mit einfach. Das liegt daran, dass wir zwei Richtungen für die Bindings haben: Die Quelle (Source), die uns die Liste der Einträge liefert, und das Ziel (Senke, Destination), in dem der ausgewählte Eintrag gesetzt werden soll. Zeichnen wir uns das zunächst einmal auf:



Partnerwahl: Die Quelle ist ein Set, die Senke ein einzelnes Objekt.

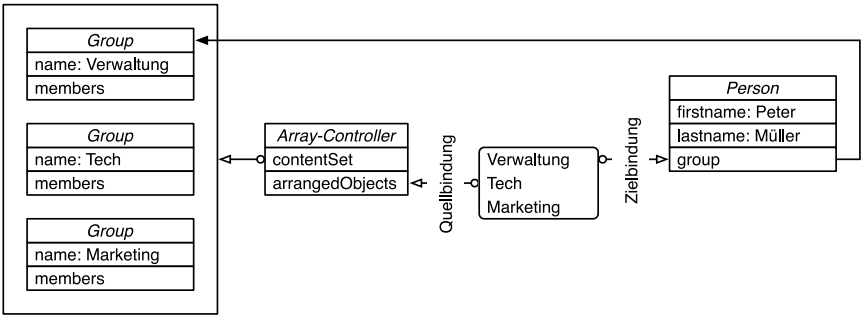
Quellbindung

Gut, versuchen wir ein Pop-up zu bauen, welches im Personenfenster die Auswahl einer Abteilung für die angezeigte Person erlaubt: In Document.xib öffnen Sie das Persons-Fenster. Schaffen Sie etwas Platz, und ziehen Sie aus der Library einen Pop-Up-Button (Suche mit: *pop*), und platzieren Sie diesen unterhalb der Tabelle, vielleicht noch daneben ein Label mit dem Text *Group*. Jetzt müssen wir dem Pop-up also eine Quelle geben. Dazu wechseln Sie in das Bindings-Pane des Inspectors und setzen dort das *Content*-Binding.

```
Bind To: Groups Controller
Controller Key: arrangedObjects
Model Key Path:
```

Wir binden also die Quelle an die *arranged Objects* des Gruppencontrollers. Das ist ja auch einsichtig. Allerdings lassen wir das Feld *Model Key Path* bewusst leer, da wir wirklich eine Auswahl der einzelnen Gruppeninstanzen haben wollen. Anders formuliert: Wir wollen einer Person nicht den Namen einer Gruppe zuweisen, sondern wirklich die Gruppe als Instanz. Da kein *Model-Key-Path* eingegeben wird, müssen Sie das Häkchen vor *Bind To:* explizit setzen.

Starten Sie jetzt das Programm und legen Sie zunächst drei Abteilungen an, wie aus der Abbildung ersichtlich. Wechseln Sie in das Personenfenster, klicken Sie auf das Pop-up. Erschrecken Sie bitte nicht. Ich weiß selbst, dass das seltsam aussieht. Wo liegt hierfür der Grund?



Binden wir nur die zu setzenden Instanzen, werden auch diese als Gesamts angezeigt.

Wir haben ja den Content an die Group-Instanzen (Abteilungen) gebunden. Dies bedeutet, dass in der Tat der Inhalt des Pop-ups nicht durch die Namen der Gruppen gebildet wird, sondern durch die Gruppeninstanzen selbst. Wir könnten jetzt unser Content-Binding so ändern, dass es auch einen Model-Key-Path auf name bekäme. Bloß würde dann eben zu der Person nicht die Abteilung gespeichert, sondern deren Name. Das passt schon nicht zu unserem Model, weil dies unter der Eigenschaft `group` eine Beziehung zu einer Group-Instanz erwartet und nicht den Gruppennamen als String.

BEISPIEL

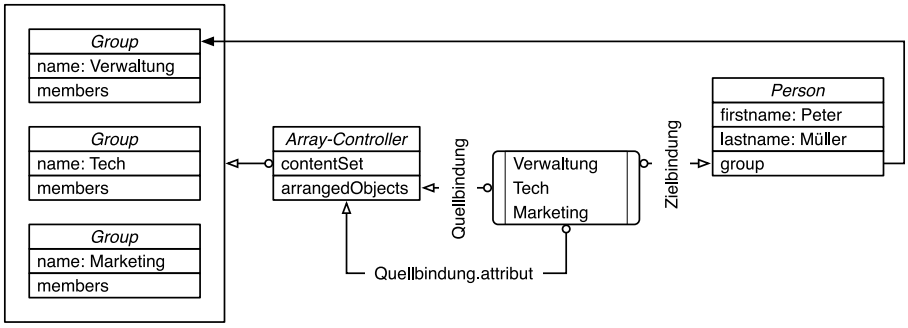
Dies kann manchmal durchaus gewollt sein! So könnte man bei unserer Converter-Anwendung daran denken, dass wir eine Entität Einheit haben, die zum Beispiel als Attribut ein Kürzel wie @"cm" speichert. Hier wäre es möglicherweise richtig, einfach den gespeicherten String zur Auswahl anzubieten. Aber bedenken Sie: Wenn wir später den String ändern, würde sich das nicht mehr auf das verweisende Objekt auswirken, da keine Beziehung besteht, sondern eine Kopie eines Attributes abgelegt wird. Will man das? Erneut Tatfrage! Hier sei lediglich die Möglichkeit erwähnt.

Langer Rede kurzer Sinn: Weil das Pop-up keinen blassen Schimmer davon hat, was angezeigt werden soll, und lediglich die verwiesenen Objekte kennt, behilft es sich etwas: Es schickt an jede der Group-Instanzen die Nachricht `description`, die ja eine Beschreibung als String zurückliefert. Diesen verwendet der Pop-up dann. Das Resultat ist dieser komische Text.

Einen Ausweg aus diesem Dilemma bietet das Content-Value-Binding. Dieses erlaubt uns, ein angezeigtes Attribut zu bestimmen, ohne dass die abgespeicherten Werte von der Instanz auf das Attribut wechseln. Beenden Sie daher das Programm, und zurück im Interface Builder wählen Sie das Pop-up an und setzen zusätzlich das Content-Value-Binding wie folgt:

```
Bind To: Groups Controller
Controller Key: arrangedObjects
Model Key Path: name
```

Für dieses Binding fügen wir also einen Model-Key hinzu, um die angezeigte Eigenschaft zu bestimmen. Wenn Sie jetzt erneut das Programm starten, bemerken Sie, dass nach der Eingabe von Abteilungen das Pop-up nachvollziehbar aussieht.



Die angezeigten Texte erhalten ein eigenes Binding.

GRUNDLAGEN

Es ist ein bisschen so wie beim Tableview, bei dem Sie eine Spalte binden: Diese zeigt dann die entsprechenden Werte an. Das Tableview selbst erhält eine Quellbindung ohne den Model-Key-Path, die Sie bloß nicht sehen. Ich hatte schon dort beschrieben, dass das überflüssig ist und eine einheitliche Bindung + einen Attributnamen deutlich einfacher wäre. Bei viewbased Tableviews hat das dann Apple auch irgendwann mal eingesehen.

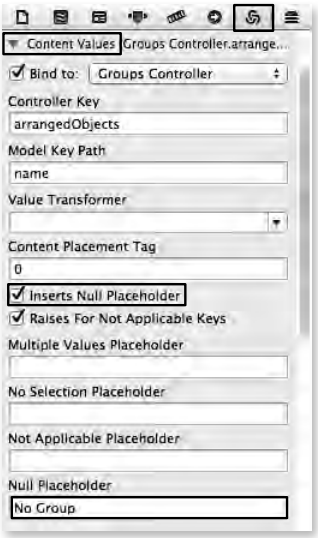
Zielbindung

Gut, das war aber nur die Quelle. Bleibt das Ziel. Wir wollen, dass das ausgewählte Objekt in der Beziehung *group* der gerade angewählten Person gespeichert wird. Dazu setzen wir beim Pop-up das *Selected Object*-Binding auf eben die Eigenschaft `group` der angezeigten Person:

```
Bind To: Persons Controller
Controller Key: selection
Model Key Path: group
```

Starten Sie das Programm. Legen Sie wieder Gruppen und sodann im Persons-Fenster eine Person an. Zunächst wundern Sie sich vielleicht, dass *No Value* erscheint. Dies ist aber zutreffend, da in der Tat kein Abteilungswert für diese Person existiert. Wo sollte der auch herkommen?

Sie können aber jetzt in dem Pop-up-Menü eine Gruppe auswählen, wenn Sie diese und eine Person eingegeben haben. Diese wird dann zu der Person gesetzt. Beachten Sie bitte auch, dass im Company-Fenster entsprechend die Listen der Personen aktualisiert werden! Bindings aktualisieren ja alles. Noch etwas: Nachdem Sie eine Gruppe ausgewählt haben, verschwindet der Eintrag *No Value*. Damit es weiterhin möglich ist, eine Person keiner Gruppe zuzuordnen, können Sie im Binding *Content Value* das Häkchen vor *Inserts Null Placeholder* setzen und unten unter *Null Placeholder* einen Text eingeben. Dann ist es ebenfalls möglich, den Verweis zu löschen.



Soll nicht nur die Auswahlliste, sondern auch eine Nicht-Auswahl erscheinen, so müssen zwei Optionen gesetzt werden.

HILFE

Sie können das Projekt in diesem Zustand als company-03 von der Webseite herunterladen.

Es geht sogar noch mehr. Setzen Sie die Eigenschaft Content Placement Tag auf einen Wert unterschiedlich von 0, so ersetzen die Gruppen aus dem Quellbinding nicht das Pop-Up-Menü des Interface Builders, sondern erweitern es. Sie können dann also ganz eigene Einträge mit der Gruppenliste kombinieren. Wir haben jedoch hierfür keine Verwendung.

Aber noch sind wir nicht ganz fertig. Denn häufig genug soll das Auswahl-Binding nicht in einem gesonderten Pop-up stehen, sondern in einer Spalte des Tableviews. Und hier gibt es einen beliebten Fehler.

Öffnen Sie noch einmal das Persons-Fenster, und fügen Sie dem Tableview unter Zuhilfenahme des Attributes-Inspectors eine dritte Spalte hinzu (*Columns*), die Sie mit *Group* bezeichnen. Auf den Text in der ersten Zeile dieser Spalte ziehen Sie aus der Library eine *Pop Up Button Cell* (Suche mit: *pop*). Jetzt wählen Sie bitte die zugehörige Spalte aus – nicht die Cell! Achten Sie auf den Titel der Navigationsleiste über dem Fenster! – Und setzen Sie das Content-Binding und das Content-Value-Binding wie beim Pop-up-Button. Beim Selected-Object-Binding müssen wir allerdings anders vorgehen: Da wir uns in einer Tabelle befinden, können wir nicht selection als Controller-Key nehmen. Denn dies würde ja nur ein Objekt liefern. Das Tableview will aber ein Array, weil es ja Zeilen hat. Daher muss hier `arrangedObjects` ausgewählt werden:

```
Bind To: Persons Controller
Controller Key: arrangedObjects
Model Key Path: group
```

Falls diese Tabellenspalte noch ein Binding *Value* haben sollte, entfernen Sie dort bitte das Häkchen vor *Bind To:*. Denn für eine Spalte mit Pop-ups ist dies nicht mehr sinnvoll und führt daher zu einem Fehler.

AUFGEPASST

Es ist wichtig, dass Sie die Bindings der Tabellenspalte setzen, da die Cell ja vom Tableview für alle Zeilen verwendet wird. Setzen Sie die Bindings der Cell, so führt dies bei der Anzeige der Auswahl und bei der Aktualisierung zu Problemen.

Erneut mit starten und testen. Geben Sie einige Abteilungen und Personen ein, ändern Sie die Bezeichnungen und die Zuordnung.

Ja, Pop-ups in Tableviews sind ein ekliges Thema. Deshalb habe ich mich nicht davor gedrückt und es hier aufgenommen. Zum Trainieren empfehle ich Ihnen, dass Sie sich

zunächst das Ganze einmal aufzeichnen. Dann sollten Sie sich ein einfaches Pop-up für die aktuelle Selektion bauen. Wenn das funktioniert, richten Sie eine Spalte im Tableview ein. Im Prinzip gehen Sie also wie bei meinen Erläuterungen vor. So kann man nach und nach testen, ob man alles richtig gemacht hat.

Natürlich können Sie jetzt wieder den Pop-Up-Button samt Label entfernen.

HILFE

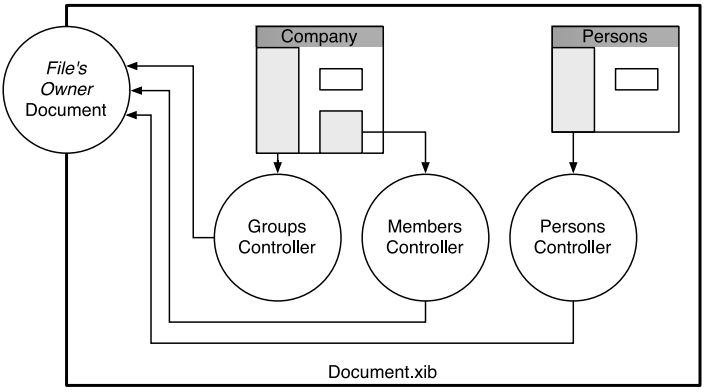
Sie können das Projekt in diesem Zustand als company-04 von der Webseite herunterladen.

6.2 Windowcontroller und Viewcontroller

Eine wichtige Aufgabe übernehmen Windowcontroller. Sie stellen ein Bindeglied zwischen einem Fenster und dem (standardmäßigen) *File's Owner* – meist einem Dokument – dar. Sie sind optional, weshalb wir bisher ohne sie auskamen. Ich kann aber nur dringend anraten, für jedes Fenster (genauer: für jede Art von Fenster) einen Windowcontroller anzubieten. Gleiches gilt für Views, wenn diese umfangreiche Funktionalität aufweisen oder separat geladen werden sollen. (Ja, so etwas machen wir hier noch.)

6.2.1 Aufgabe und Stellung

Ich weise mal auf die »Missstände« unserer bisherigen Arbeit hin: Wenn wir uns die verschiedenen Beziehungen im Nib der Applikation anschauen, entsteht ein ziemliches Geflecht.



In Nibs entstehen schnell unübersichtliche Strukturen.