LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Praktikum iOS-Entwicklung

Sommersemester 2015

Prof. Dr. Linnhoff-Popien

Florian Dorfmeister, Mirco Schönfeld

Gemeinsames Themen-Brainstorming am 3.6. – also schon nächste Woche!

Wir suchen Ideen für die Praxisphase

Das heißt:

- Eure Ideen sind gefragt!
- Vorstellen der Ideen in 5-minütigen Präsentationen
- Vergabe der Themen mit Beginn der Programmierphase

Blöcke und Variablen Capturing in Objective-C

Was ist ein Block?

Ein Block (Closure) ist eine zusammenhängende Abfolge von Befehlen, die zu einem späteren Zeitpunkt ausgeführt wird

Ein Block hat wie eine C-Funktion Argumente und liefert einen Rückgabewert

Ein Block ist aber gleichzeitig ein Objekt, das an Variablen gebunden und weitergegeben werden kann.

Beispiel:

- Block als Callback in `dismissViewControllerAnimated:completion:` (Methode der Klasse `UIViewController`):

```
[self dismissViewControllerAnimated:YES
  completion:^{
    [self performSomeFinalActions];
  };
]
```

Deklaration, Initialisierung und Verwendung von Blöcken

Man verwendet den ^-Operator, um ...

- ... eine Block-Variable zu deklarieren
- ... ein Block-Literal zu kennzeichnen

Blöcke sind als sog. Closure implementiert

- Blöcke können Variablen benutzen, die sich im Gültigkeitsbereich (*Enclosing Scope*) befinden, in dem der Block selbst definiert wurde (hier `factor`)

Beispiel:

```
int factor = 2;

int (^doubleMe)(int) = ^int(int number) {
   return number * factor;
};
```

Deklaration, Initialisierung und Verwendung von Blöcken

Erläuterung

Deklaration der Variablen `doubleMe;`

„`^`“ Kennzeichnet die Variable als Block

Literale Block-Definition, die der Variablen `doubleMe` zugewiesen wird

```
int (^doubleMe)(int) = ^int(int number) {return number * factor;};
```

`doubleMe` ist ein Block der `int` zurück liefert

Das einzige Argument ist vom Typ `int`

Optional: Typ des Rückgabe werts

Argument mit dem Namen `number`

Funktionsrumpf des Blocks

Deklaration, Initialisierung und Verwendung von Blöcken

Man kann Deklaration und Initialisierung von Blöcke natürlich auch trennen

Beispiel

```
int factor = 2;

int (^doubleMe)(int);

doubleMe = ^(int number) {
   return number * factor;
};
```

Deklaration, Initialisierung und Verwendung von Blöcken

Blöcke ohne Parameter müssen in der Definition als solche gekennzeichnet werden

In der Initialisierung sind Abkürzungen erlaubt

(Ein etwas sinnfreies) Beispiel für valide Block-Initialisierungen

```
void(^foo)(void) = ^void(void){ /*bar*/ };
// oder
void(^foo)() = ^void(void){ /*bar*/ };

// auch OK sind...
void(^foo)() = ^(void){ /*bar*/ };
void(^foo)() = ^(){ /*bar*/ };
void(^foo)() = ^{ /*bar*/ };
```

Deklaration, Initialisierung und Verwendung von Blöcken

Blöcke, die als Variable deklariert wurden, können wie eine Funktion verwendet werden

Beispiel:

```
int factor = 2;
int (^doubleMe)(int) = ^(int number) {
    return number * factor;
};

printf("%d", doubleMe(3)); // Ausgabe "6"
```

Verwendung von Blöcken in Cocoa

Blöcke werden oft als Argument von Methoden verwendet, um ...

- ... eine Operation auf eine Menge von Objekten anzuwenden
- ... um ihn als Callback nach Beendigung einer Operation auszuführen

Blöcke können auch *inline* verwendet werden

```
int main(int argc, char * argv[]) {
   NSArray *myArray =
      @[@"This", @"will", @"be", @"printed", @"and", @"this", @"won't"];

   [myArray enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
      NSLog(@"Entry %d: %@", idx, obj);
      if([@"printed" isEqualToString:obj]){
         *stop = YES;
      }
   }
   ];
}

// Ausgabe
Entry 0: This
Entry 1: will
Entry 2: be
Entry 3: printed
```

Verwendung von Blöcken in Cocoa

Mit Blöcken können Funktionalitäten von Objekten schnell ausgetauscht werden.

Beispiel: `MyEquationComputer`

```
// MyEquationComputer.h

@interface MyEquationComputer : NSObject
@property (nonatomic, copy) int (^equation)(int, int); // copy!!
- (int)computeWithValue:(int)value1 andValue:(int)value2;
@end
```

```
// MyEquationComputer.m

@implementation MyEquationComputer
- (int)computeWithValue:(int)value1 andValue:(int)value2 {
    if(!self.equation) return 0; // Der Block wurde noch nicht Initialisiert
    return self.equation(value1, value2);
}
@end
```

Beispiel: MyEquationComputer

```
// Verwendung (z.B. in einem UIViewController)

MyEquationComputer *mec = [MyEquationComputer new];

[mec setEquation:^int(int a, int b){ return a+b; }];
int sum = [mec computeWithValue:5 andValue:3];
NSLog(@"Sum = %d", sum); // Sum = 8

[mec setEquation:^int(int a, int b){ return a*b; }];
int prod = [mec computeWithValue:5 andValue:3];
NSLog(@"Prod = %d", prod); // Prod = 15

int multiplier = 3;
[mec setEquation:^int(int a, int b){ return (a+b) * multiplier; }];
sum = [mec computeWithValue:5 andValue:3];
NSLog(@"Sum = %d", sum); // Sum = 24

multiplier = 4;
sum = [mec computeWithValue:5 andValue:3];
NSLog(@"Sum = %d", sum); // Sum = 24 (Warum?)
```

Verwendung von Blöcken in Cocoa

Wenn man einen Block als Property deklariert, muss man das Schlüsselwort copy verwenden!

- Blöcke werden auf dem Stack abgelegt (nicht wie Objekte auf dem Heap!)
- Blöcke, die innerhalb einer Methode deklariert wurden, werden (wie alle anderen lokalen Variablen der Methode) bei deren Rückkehr gelöscht.

Mit copy wird eine Kopie des Block auf dem Heap erzeugt

- Die Kopie bleibt somit auch über den Methodenaufruf hinweg erhalten

Variablen Capturing

Ein Block kann auf alle Variablen zugreifen, die sich im selben Gültigkeitsbereich (Enclosing Scope) des Blocks befinden

Ein Block hat Zugriff auf

- lokale Variablen einer Methode
- Übergabeparameter einer Methode
- alle Instanzvariablen, die zu dem Objekt gehören, auf welches die entsprechende Methode aufgerufen wurde

Diese Variablen werden auch als **Captured Variablen** bezeichnet

Variablen Capturing

Achtung!

- Blöcke halten starke Referenzen zu allen Objekten, die sie verwenden.
- Es besteht die Gefahr zur Bildung von Speicherzyklen (strong reference cycles)
- Beispiel:

```
@property (nonatomic, strong) NSArray *myArr;

// In der Implementierung:
[self.myArr addObject:^() {
  [self createSomeObject];
}];
```

- Es kommt zu einem Memory Cycle:
  - Block hält eine Referenz zu `self`
  - `self` hält einen starken Zeiger auf den Block

→Weder der Block noch die Instanz können gelöscht werden!

Variablen Capturing

Lösung:

- Deklaration schwacher Referenz auf entsprechende Capture Variablen
- Ausschließliche Verwendung der schwachen Referenz innerhalb des Blocks

```
@property (nonatomic, strong) NSArray *myArr;

// In der Implementierung:
__weak MyClass *weakSelf = self;

[self.myArr addObject:^() {
  [weakSelf doSomething];
}];
```

Variablen Capturing

Beispiel:

- Speichern der Historie von `MyEquationComputer`

```
// MyEquationComputer.h
[..]
@property (nonatomic, strong) NSMutableArray* history;
[...]
```

```
// Verwendung (z.B. in einem UIViewController)

[...]
 __weak MyEquationComputer *weakMec = mec;

    [mec setEquation:^int(int a, int b){
        int result = a + b;
        [weakMec.history addObject:[NSNumber numberWithInt:result]];
        return result;
    }];

[...]
```

Variablen Capturing

Der Zugriff auf Captured Variablen ist READ-ONLY!

READ-WRITE Zugriff mit Hilfe des `__block` Speichertyps

Beispiel:

```
__block int x = 123; //  x existiert im __block Speicher

void (^printXAndY)(int) = ^(int y) {
    x = x + y;
    printf("%d %d\n", x, y);
};

printXAndY(456); // Ausgabe: 579 456
// Achtung! Nach dem ersten Aufruf gilt: x = 579
// Wert von x nach weiteren Aufrufen?
```

Verwendung von Blocks in iOS

Enumeration

Animation

Sortieren

- Block definiert Vergleichsoperation

Notifications

- Block definiert Reaktion auf Notification

Error Handler

- Block definiert Reaktion auf Fehler

Completion Handler

- Wenn etwas fertig ist, führe Block aus
- Beispiel: Asynchrone Kommunikation

Closures und Variablen Capturing in Swift

Was sind Closures

Closures in Swift ähneln Blöcken in Objective-C

- Closures können auf jegliche Art von Referenzen zugreifen, die sich in dem Kontext befinden, in dem sie definiert wurden (Enclosing Scope)

- Closures erscheinen als
  - globale Funktionen mit Namen und ohne Captured Variablen
  - eingebettete Funktionen mit Namen: Captured Variablen sind solche, die in der sie einbettenden Funktion auftreten
  - Closure Ausdruck ohne Namen: Captured Variablen sind solche, die sich in ihrem umgebenden Kontext befinden

Closures als globale Funktion

Beispiel:

- Basierend auf dem Rückgabewert eines vorgegebenen Closures sortiert die Funktion `sorted` ein Array von Werten bekannten Typs
- Für das Sortieren einer Liste von Strings benötigt man ein Closure vom Typ `(String, String) -> Bool`

```
let words = ["this", "will", "be", "printed"]

func sortBackwards(s1: String, s2: String) -> Bool  {
    return s1 > s2
}

var reversed = sorted(words, sortBackwards)
// Ergebnis: ["will", "this", "printed", "be"]
```

Inline Closures

Closures lassen sich ähnlich zu Blöcken in Objective-C auch in inline Notation verwenden

Syntax:

```
{( <Parameter Liste> ) -> <Datentyp des Rückgabewerts> in

   <Ausdrücke>

}
```

Beispiel:

```
reversed = sorted(words, { (s1: String, s2: String) -> Bool in
   return s1 > s2})
```

Inline Closures

Abkürzende Schreibweisen:

```
reversed = sorted(words, { s1, s2 in return s1 > s2 } )
// Der Typ String erschließt sich aus dem Kontext
// (Aufruf von sorted mit Array vom Typ [String])

reversed = sorted(words, { s1, s2 in s1 > s2 } )
// Bei einzeiligen Ausdrücken ist klar, was der Rückgabewert ist
// (Schlüsslewort return kann weggelassen werden)
```

Inline Closures

Shorthand Argumente:

- Swift weist jedem Argument(in der Reihenfolge seines Auftretens) automatisch einen Shorthand-Namen zu (`$0, $1, $2, ...`)
- Bei Verwendung von Shorthands kann man die Liste der Argumente weglassen
- Beispiel:

```
reversed = sorted(words, { $0 > $1 } )
```

Weitere Informationen zur Verwendung von Closures im *"Swift Programming Language Guide"*

Variablen Capturing

Das Variablen Capturing funktioniert analog zu Objective-C

Beispiel:

- Funktion zum Inkrementieren einer Variablen um einen statischen Wert

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var result = 0
    func incrementor() -> Int {
        result += amount
        return result
    }
    return incrementor
}

let incrementByTen = makeIncrementor(forIncrement: 10)

incrementByTen()
// result = 10
incrementByTen()
// result = 20
```

Closures Speichermanagement

- Wie in Objective-C können Closures auch der Property einer Klasse zugewiesen werden
- Wenn innerhalb des Closures auf ein Instanz (oder Instanzvariablen) der Klasse zugegriffen wird, entsteht ein Speicherzyklus (strong reference cycle) zwischen dem Closure und der Instanz!
- Swift verwaltet Capture-Listen, um solche Speicherzyklen zu durchbrechen und zu verwalten

→ In Swift ist keine gesonderte Behandlung notwendig (kein Anlegen einer Kopie vom Typ `__weak` o.ä.)

Collection Views

Collection Views: Überblick

Beispiel:

Quelle: http://www.raywenderlich.com

Collection Views: Überblick

Collection Views...

- ... dienen der Visualisierung einer geordneten Menge von Elementen
- ... erlauben den Einsatz eines flexiblen und veränderbaren Layouts

Meistens erfolgt die Darstellung der Inhalte in einem **gitterähnlichen Layout**

Durch **Vererbung** kann sehr individuell auf das Layout Einfluss genommen werden

- Grids
- Stacks
- Zirkuläre Layouts
- Dynamisch adaptive Layouts
- ...

Collection Views: Überblick

Collection Views **trennen strikt** zwischen...

- ... den **präsentierten Daten** und
- ... den **UI-Elementen**, die zur Präsentation der Daten verwendet werden (MVC!)

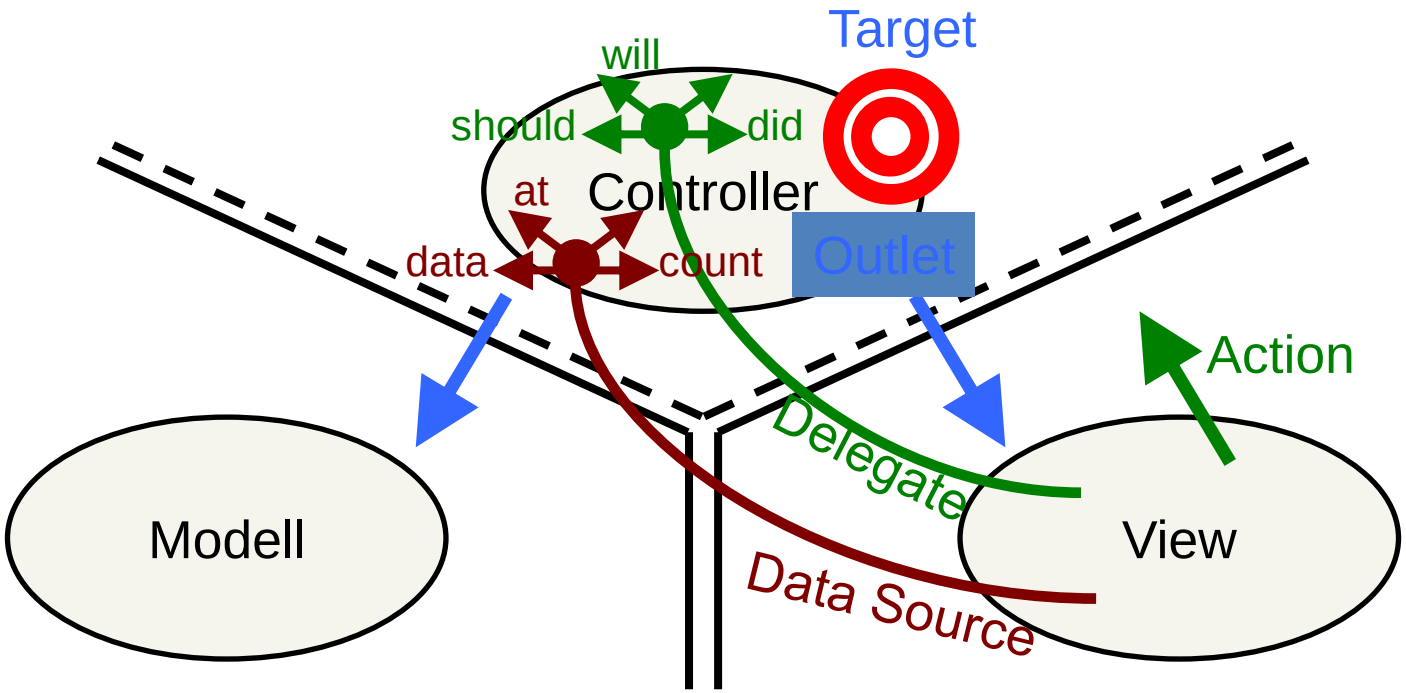
Die **Anwendung** (der Controller) kümmert sich um die...

- ... Bereitstellung der **Daten**
- ... Bereitstellung der einzelnen **View-Objekte**

Die Collection View kümmert sich um **die Darstellung** (Positionierung, Layout) der View-Objekte

Collection Views besitzen ein gesondertes **Layout-Objekt** zur Darstellung der View-Objekte

Collection Views: MVC



Collection Views ...

- ... benötigen eine Data Source (z.B. zur Ermittlung der Anzahl darzustellender Objekte)
- ... können optional mit Delegates kommunizieren (zur Auslieferung „interessanter Nachrichten“ wie z.B. die Selektion von Zellen durch den Nutzer)

Collection Views: Kollaborierende Objekte

Die visuelle Repräsentation von Collection Views erfolgt durch zahlreiche unterschiedliche Objekte

Die meisten dieser Objekte können so verwendet werden, wie sie UIKit zur Verfügung stellt

Vererbung ist meistens nur dann notwendig, wenn eine besondere Funktionalität erwünscht ist

Collection Views: Kollaborierende Objekte

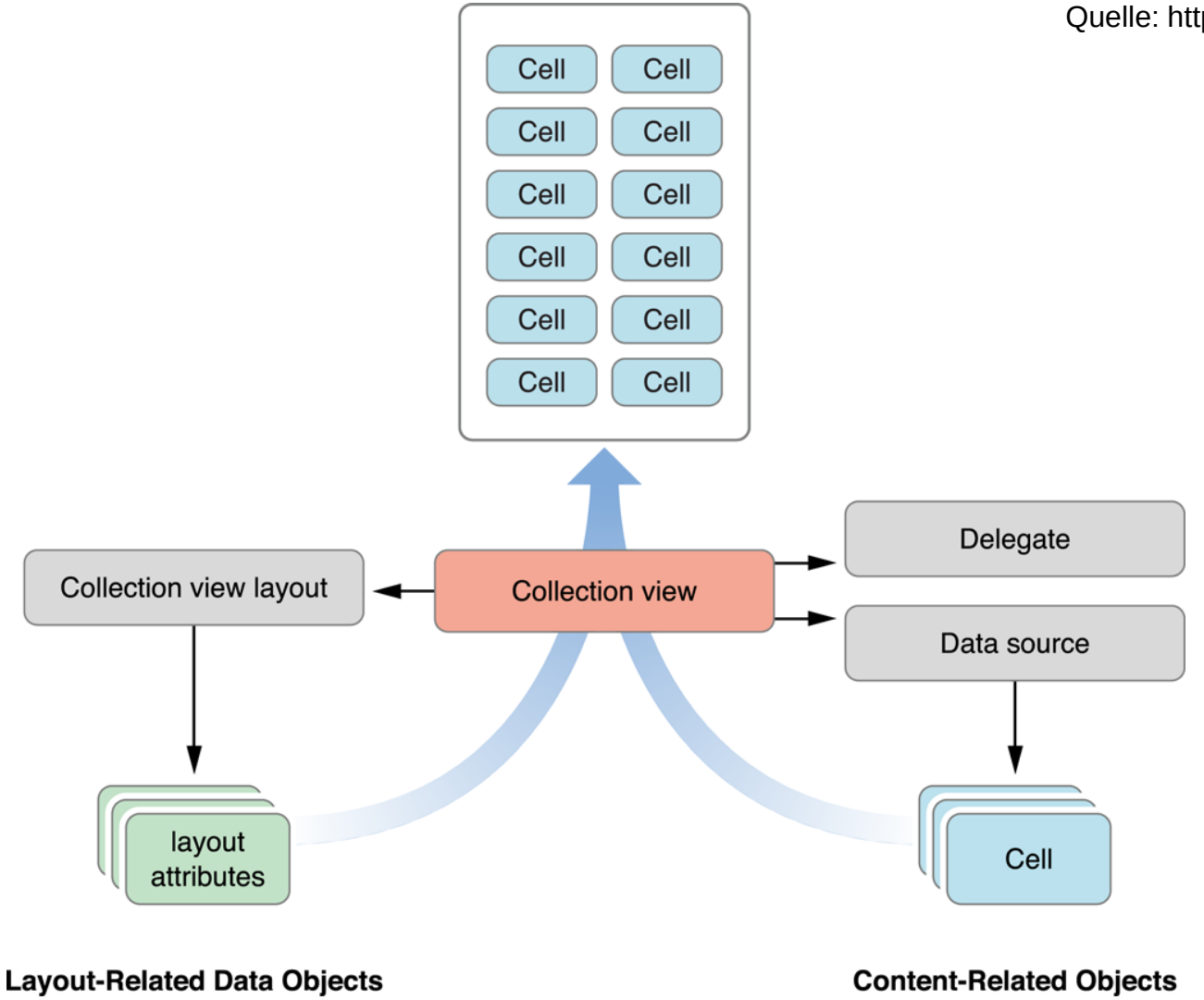
Zweck	Klasse / Protokoll	Beschreibung
Top-Level Containment / Management	`UICollectionView`	• Die sichtbare Fläche der Collection View Inhalte • Erbt von `UIScrollView` • Darstellung erfolgt auf der Basis des assoziierten Layout-Objekts • Muss nicht notwendigerweise die gesamte Anzeige ausfüllen
	`UICollectionViewController`	• **Optional** • Bietet Unterstützung für das View Controller Management
Content Management	`UICollectionViewDataSource`	• **Required** • Regelt die Darstellung des Inhalts der Collection View • Erzeugt benötigte Views auf Anforderung der Collection View
	`UICollectionViewDelegate`	• **Optional** • Ermöglicht das Empfangen interessanter Nachrichten von der assoziierten Collection View • Dient zur Anpassung des Verhaltens der View • z.B. Verfolgen von Auswahl / Markierung von Zellen
Präsentation	`UICollectionReusableView`	• Präsentierte Views innerhalb einer Collection View müssen Instanzen dieser Klasse sein • Implementiert einen Recycling-Mechanismus
	`UICollectionViewCell`	• Objekte dieser Klasse besitzen einen spezielle Recycling Typ (erbt von `UICollectionReusableView`) • Wird für die Hauptinhalte verwendet

Collection Views: Kollaborierende Objekte

Zweck	Klasse / Protokoll	Beschreibung
Layout	`UICollectionViewLayout`	• Zur Definition der Position, Größe und visueller Attribute einer Zelle / wiederverwendbarer (reusable) Objekte
	`UICollectionViewLayoutAttributes`	• Werden während des Layout-Prozesses erzeugt • Liefern der Collection View die Information darüber, wo und wie Zellen / wiederverwendbarer Objekte dargestellt werden müssen
	`UICollectionViewUpdateItem`	• Layout Objekt erhält Instanzen dieser Klasse, wenn Inhalte innerhalb der Collection View eingefügt, verschoben oder gelöscht werden • Instanzen dieser Klasse muss man nie selbst erzeugen!
Flow Layout	`UICollectionViewFlowLayout`	• Konkretes Layout, das Apple zur Verfügung stellt • Wird zur Implementierung von Grids und zeilenbasierten Layouts verwendet • Kann direkt mit der Standard-Implementierung verwendet werden • Kann zur dynamischen Adaption der Layout Information in Verbindung mit einem FlowLayout Delegate Objekt verwendet werden
	`UICollectionViewDelegateFlowLayout`	• Das mit dem `UICollectionViewFlowLayout` Objekt assoziierte Delegate Objekt • Methoden dieses Protokolls liefern die Größe von und die Abstände zwischen den Items

Collection Views: Kollaborierende Objekte

Quelle: http://www.apple.com/



Collection Views: Reusable Views

Views, die von der Anzeige „verschwinden“ werden nicht zerstört, sondern in einer Reuse-Queue gespeichert

Sobald neue Inhalte angezeigt werden müssen, wird eine View ...

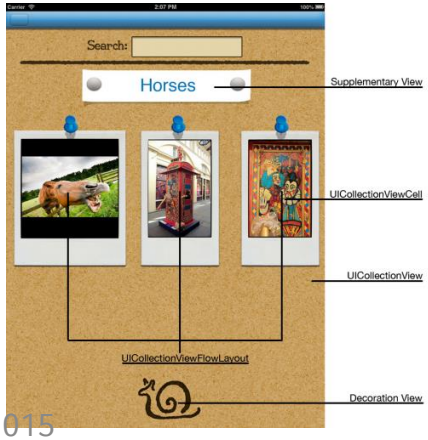
- ... aus der Reuse-Queue geholt,
- ... mit den entsprechenden Inhalten konfiguriert
- ... und angezeigt

Damit das Recycling funktioniert müssen alle Views der Collection View von der Klasse `UICollectionReusableView` erben

Collection Views: Reusable Views

Es werden drei Arten wiederverwendbarer Objekte unterschieden

- Zellen
  - Sind Instanzen der Klasse `UICollectionViewCell`
  - Dienen zur Darstellung eines einzelnen Items der Data Source
  - Besitzen einen inhärenten Mechanismus zur Darstellung ihres Zustandes (selected, highlighted)
- Supplementary Views
  - **Optional**
  - Dienen der Darstellung von Informationen einer Sektion
  - Darstellung und Platzierung wird durch das Layout Objekt gesteuert
  - Beispiel Flow Layout:
    - Unterstützt Header- und Footer-Sektionen
- Decoration Views
  - Dienen ausschließlich zur Verzierung
  - Sind nur an das Layout-Objekt gebunden



Collection Views: Visuelle Repräsentation

Das Layout-Objekt ...

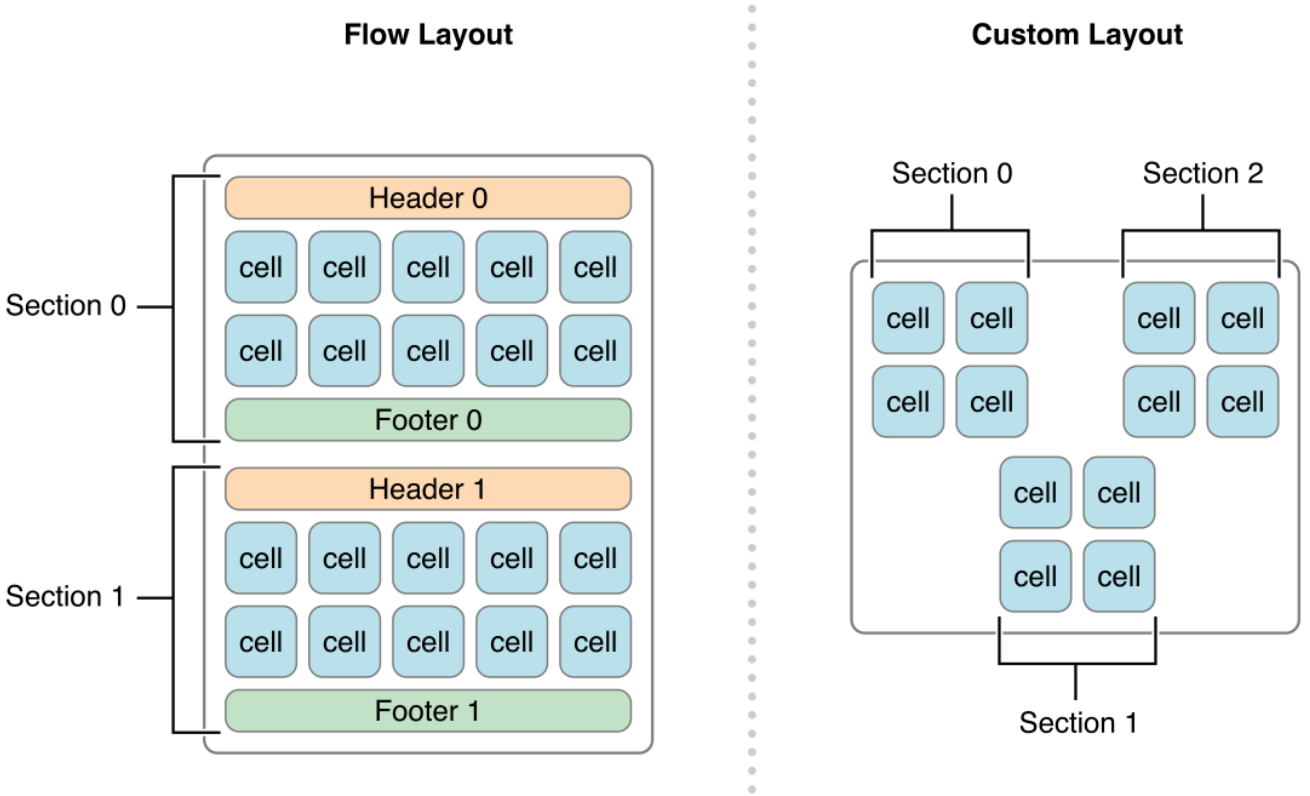
- ... dient ausschließlich der visuellen Repräsentation
- ... bestimmt die Größe, Position und andere mit der Darstellung assoziierte Attribute der Views (Transparenz, Transformationen im 3D-Raum, ...)
- ... ermöglicht die dynamische Anpassung der Darstellung einer Collection View
- ... verändert **niemals direkt** die Inhalte der dargestellten Views
- ... erzeugt Attribute (Instanzen der Klasse `UICollectionViewLayoutAttributes`), die den Ort, die Größe und die visuelle Repräsentation von Zellen, Suplementary Views und Decoration Views beschreiben

Es ist die Aufgabe der Collection View, die vom Layout-Objekt definierten Attribute auf die Views anzuwenden

Collection Views: Visuelle Repräsentation

Beispiel: Flow Layout und Custom Layout:

- Die Größe der Zellen und Supplementary Views werden als Eigenschaften des Layout-Objekts oder des Delegates spezifiziert



Quelle: http://www.apple.com/

Collection Views: Animationen

Collection Views...

- ... animieren automatisch das Einfügen, Entfernen, ... von Items oder Sektionen
- ... animieren automatisch die Neuplatzierung aller betroffenen Items
- ... erlauben das Invalidieren eines Layouts.
  - Manuelles Invalidieren erfordert Implementierung eigener Animationen

Collection Views: Data Source

Das Data Source Objekt ...

- ... bietet Zugriff auf die Inhalte, welche die Collection View präsentiert
- ... ist ein Objekt des Datenmodells oder der View Controller, der die Collection View managt
- ... muss das `UICollectionViewDataSource` Protokoll implementieren

Die dargestellten Inhalte werden in Sektionen und Items unterteilt

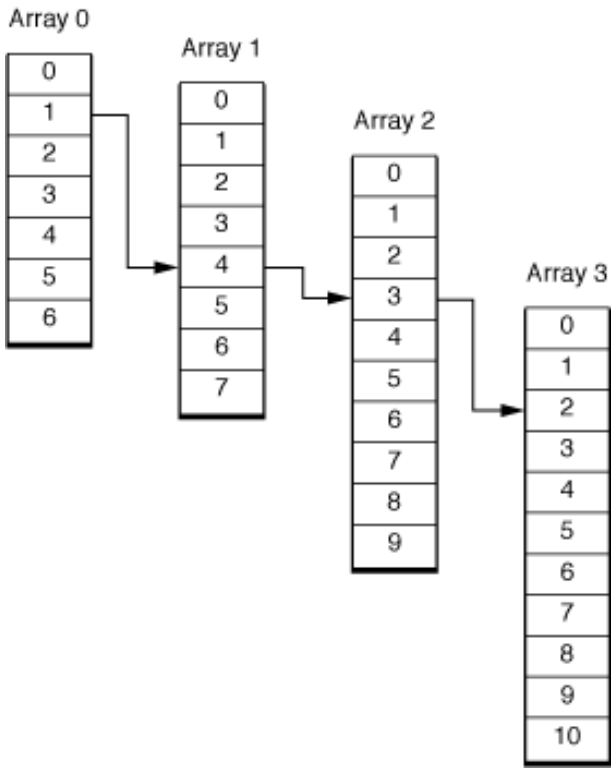
- Beispiel: Foto App mit mehreren Fotos (Items) pro Album (Sektion)

Items werden über `NSIndexPath`-Objekte referenziert

Einschub: NSIndexPath

`NSIndexPath`-Objekte repräsentieren den Pfad zu einem Knoten eines Baumes oder eines verschachtelten Arrays

Beispiel: Pfad zum Element mit dem Index 1.4.3.2.



Quelle: http://www.apple.com/

Einschub: NSIndexPath

`NSIndexPath` bietet zahlreiche Methoden zur Initialisierung und Auswertung von `NSIndexPath`- Objekten

Erzeugung:

- `+indexPathWithIndex:`
- `+indexPathWithIndexes:length:`
- `-initWithIndex:`
- `-initWithIndexes:length:`
- `-init`

Auswertung:

- `-indexAtPosition:`
- `-indexPathByAddingIndex:`
- `-indexPathByRemovingLastIndex`
- `-length`
- `-getIndexes:`

Einschub: NSIndexPath

iOS bietet eine Category zu `NSIndexPath`, um Sektionen bzw. Items innerhalb von `UITableView`-Objekten zu referenzieren

Diese werden auch bei der Programmierung mit `UICollectionViews` verwendet

- Auszug aus `UITableView.h`

```
// This category provides convenience methods to make it easier to use
// an NSIndexPath to represent a section and row
@interface NSIndexPath (UITableView)
+ (NSIndexPath *)indexPathForRow:(NSInteger)row inSection:(NSInteger)section;
@property(nonatomic,readonly) NSInteger section;
@property(nonatomic,readonly) NSInteger row;
@end
```

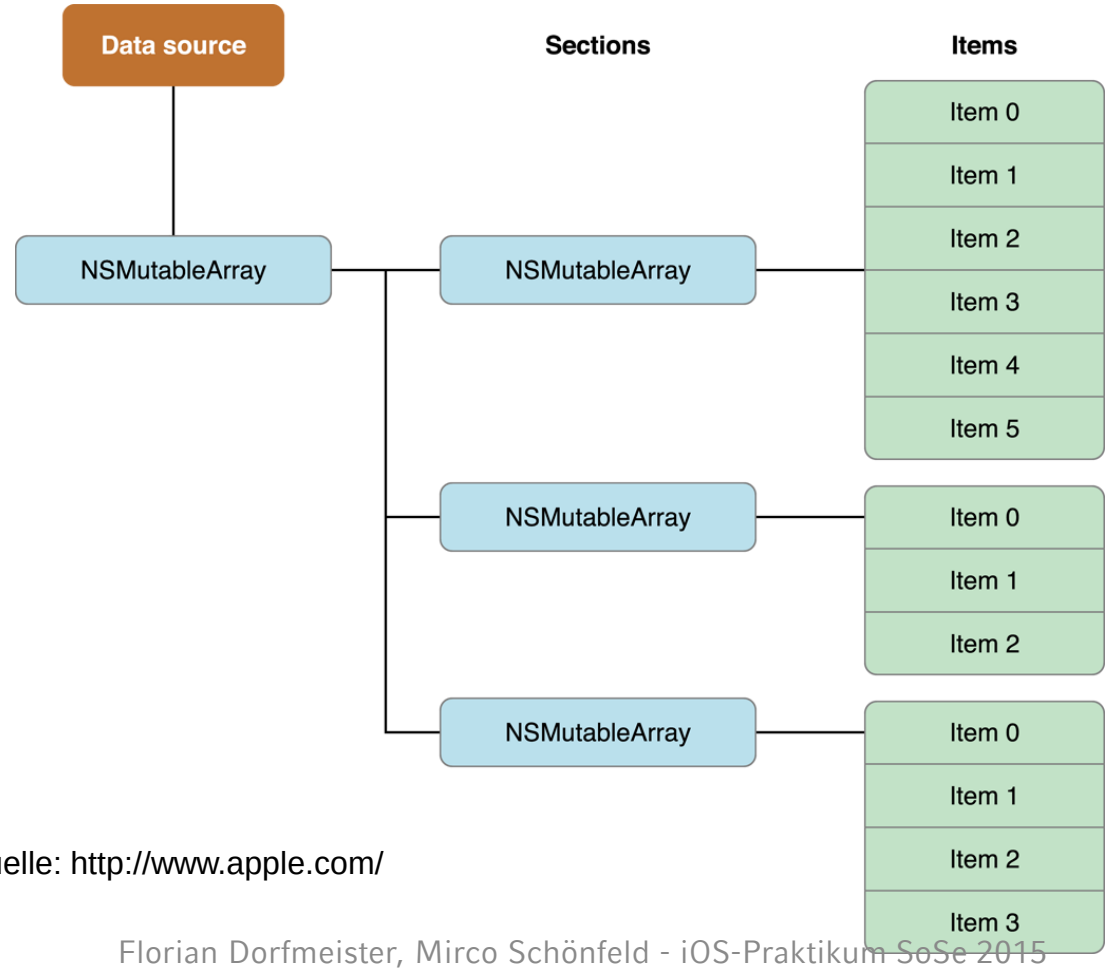
- Auszug aus `UICollectionView.h`

```
@interface NSIndexPath (UICollectionViewAdditions)
+ (NSIndexPath *)indexPathForItem:(NSInteger)item
    inSection:(NSInteger)section NS_AVAILABLE_IOS(6_0);
@property (nonatomic, readonly) NSInteger item NS_AVAILABLE_IOS(6_0);
@end
```

Collection Views: Data Source

Einfach Möglichkeit zur Realisierung einer Data Source:

- Verwaltung von Sektionen und Items als verschachteltes Array



Quelle: http://www.apple.com/

Collection Views: Protokolle

Das `UICollectionViewDataSource` Protokoll dient der Collection View dazu, Informationen von der Data Source zu erfragen

Erfragen der Anzahl an Items in einer Sektion (**required**)

- `collectionView:numberOfItemsInSection:`

Erfragen der Anzahl an Sektionen (**optional**, 1 als Standardannahme):

- `numberOfSectionsInCollectionView:`

Collection Views: Protokolle

Die Aufrufe der Methoden des `UICollectionViewDataSource`-Protokolls erfolgen, falls ...

- ... die Collection View das erste Mal angezeigt wird
- ... der Collection View ein neues Data Source Objekt zugewiesen wird
- ... explizit der Aufruf `reloadData` (der Klasse `UICollectionView`) erfolgt
- ... das Delegate einen Block durch Aufruf der Methode `performBatchUpdates:completion:` ausführt
- ... das Delegate eine der `move`-, `insert`- oder `delete`-Methoden aufruft

Collection Views: Protokolle

Beispiel:

- `self.data` stellt das Top-Level `NSArray`-Objekt dar (Sektionen)
- Eines der Array-Elemente referenziert jeweils ein `NSArray`-Objekt mit den Items der entsprechenden Sektion

```
-(NSInteger)numberOfSectionsInCollectionView:
        (UICollectionView*)collectionView {
    // self.data ist eine Property, die ein NSArray-Objekt pro Sektion
    // referenziert
    return [self.data count];
}

-(NSInteger)collectionView:(UICollectionView*)collectionView
    numberOfItemsInSection:(NSInteger)section {
    NSArray* sectionArray = [self.data objectAtIndex:section];
    return [sectionArray count];
}
```

Collection Views: Zellen und Supplementary Views konfigurieren

Eine Collection View ...

- ... verfolgt **nicht** die Veränderungen der dargestellten Inhalte
- ... stellt lediglich die übergebenen Views entsprechend des assoziierten Layout-Objekts dar

Die Darstellung der Views liegt in der Verantwortung der App/des Controllers

Eine Collection View ...

- ... frägt beim Layout-Objekt nach, wie die Inhalte dargestellt werden sollen
- ... geht davon aus, dass sie für die Erzeugung der Zellen / Supplementary Views verantwortlich ist
- ... verwendet Reuse-Queues und Reuse Identifier, um Zellen / Supplementary Views effizient zur Verfügung zu stellen

Collection Views: Zellen und Supplementary Views registrieren

Registrieren von Zellen (bzw. Supplementary Views) über Storyboards:

- Einfügen einer `UICollectionViewCell` (bzw. einer `UICollectionReusableView`) von der Objekt-Library in der entsprechenden Collection View
- Setzen der **Custom Class** des `UICollectionViewCell`-Objekts (bzw. des `UICollectionReusableView`-Objekts)
- Setzen **eines Reusable View Identifiers** für das `UICollectionViewCell`-Objekt (bzw. das `UICollectionReusableView`-Objekt)

Collection Views: Zellen und Supplementary Views registrieren

Programmatisches Registrieren von Zellen:

- Assoziieren der View mir einem **Reusable View Identifier** über
  - `registerClass:forCellWithReuseIdentifier:`
  - `registerNib:forCellWithReuseIdentifier:`

Programmatisches Registrieren Supplementary Views:

- Assoziieren der View mir einem **Reusable View Identifier** über
  - `registerClass:forSupplementaryViewOfKind:withReuseIdentifier:`
  - `registerNib:forSupplementaryViewOfKind:withReuseIdentifier:`
- Der **Kind-String** definiert die Art der Supplementary View
  - Beispiel für den Kind String bei `UICollectionViewFlowLayout`:
    - `UICollectionElementKindSectionHeader`
    - `UICollectionElementKindSectionFooter`

Aufruf der Methoden erfolgt z.B. während des Initialisierungsprozesses im zugehörigen View Controller

Dequeueing und Konfigurieren von Zellen und Supplementary Views

- Das Data Source Objekt ist verantwortlich für die Bereitstellung der Inhalte und Konfiguration der Zellen und Supplementary Views

- Dazu Implementierung der `UICollectionViewDataSource`-Protokollmethoden
  - `collectionView:cellForItemAtIndexPath:`
    - **Required**
  - `collectionView:viewForSupplementaryElementOfKind:atIndexPath:`
    - **Optional**: Hängt vom verwendeten Layout ab

- Methoden müssen für das gegebene `NSIndexPath`-Objekt eine entsprechende View zurückliefern

Collection Views: Zellen und Supplementary Views konfigurieren

Die Implementierung der Methoden erfolgt nach einem einheitlichen Muster

- Dequeueing einer Zelle bzw. View des entsprechenden Typs durch Aufruf einer der Methoden
  - `dequeueReusableCellWithReuseIdentifier:forIndexPath:`
  - `dequeueReusableSupplementaryViewOfKind: withReuseIdentifier:forIndexPath:`
- Konfiguration der View mit den Daten für das gegebene `NSIndexPath`-Objekt
- Rückgabe der View

Collection Views: Zellen und Supplementary Views konfigurieren

Solange vor dem ersten Aufruf einer Dequeueing-Methode eine Zelle bzw. View registriert wurde, liefert die Methode **niemals** `nil` zurück

Falls beim Aufruf einer Dequeueing-Methode noch keine Zelle oder View in der Queue existiert, erzeugt die Methode ein entsprechendes Objekt

- aus dem Storyboard
- aus der registrierten Nib-Datei
- über den Aufruf der `initWithFrame:`-Methode der registrierten Klasse

Falls beim Aufruf einer Dequeueing-Methode bereits eine Zelle oder View existiert, erfolgt der Aufruf der `prepareForReuse`-Methode

- Kann **optional** zum Zurücksetzen einer Custom Cell View überschrieben werden

Collection Views: Zellen und Supplementary Views konfigurieren

Beispiel:

- Konfiguration der entsprechenden Zelle mit dem Index seiner Sektion und seines Items als Label

```
-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
                 cellForItemAtIndexPath:(NSIndexPath *)indexPath {

    // Holen einer Reusable View
    MyCustomCell* newCell = [self.collectionView
        dequeueReusableCellWithReuseIdentifier:MyCellID
                                  forIndexPath:indexPath];

    // Konfigurieren der View
    newCell.cellLabel.text = [NSString stringWithFormat:
        @"Section:%d, Item:%d", indexPath.section, indexPath.item];

   return newCell;
}
```

Collection Views: Weiterführende Themen

Zahlreiche weitere Möglichkeiten zur Verwendung von Collection Views werden im *"Collection View Programming Guide for iOS"* beschrieben

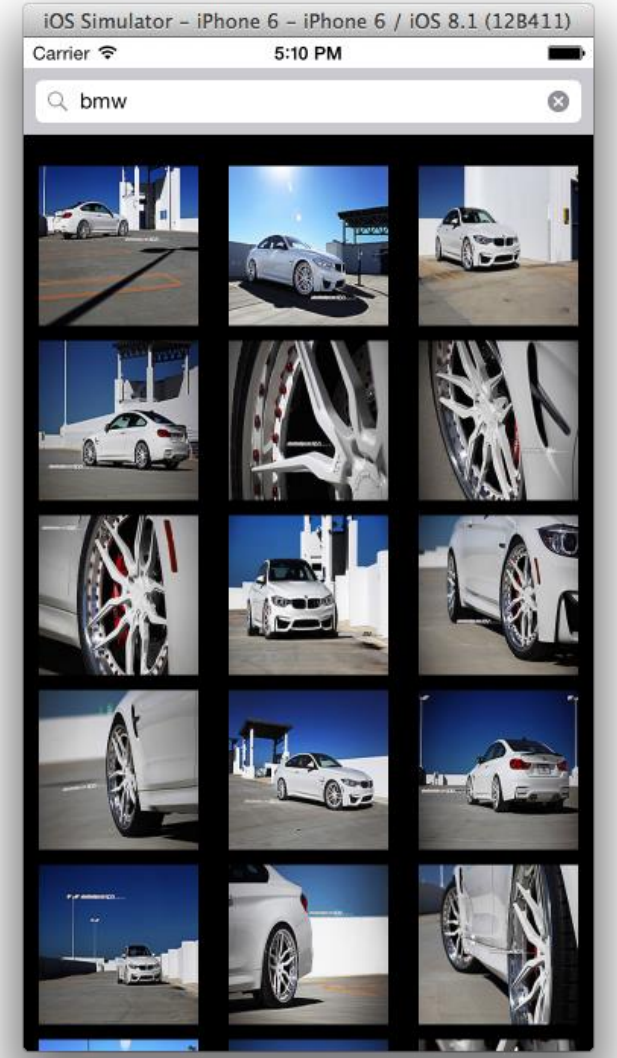
- Hinzufügen, Entfernen und Verschieben von Sektionen / Items
- Visualisierung und Behandlung von Auswahl- und Highlight-Zuständen
- Anzeigen eines Edit-Menüs für einzelne Zellen
- Transitionen zwischen Layouts
- Verwenden von Apples Flow Layout
- Integration von Gesten
- Erzeugen von Custom Layouts
- ...

Collection Views: Beispiel

FlickrSearch App

- App ermöglicht das Suchen nach Bildern auf Flickr, die auf einen bestimmten Suchbegriff passen
- Für jedes Ergebnis erzeugt die App eine neue Sektion mit den auf den Suchbegriff passenden Bildern als Items

Erzeugen des Projekts FlickrSearch
(Template: Single View Application)



Collection Views: Beispiel

Design der UI:

- Im Storyboard:
  - Hinzufügen einer `UISearchBar`
  - Anpassen von Attributen im Attributes Inspector
    - Placeholder = "Search term"
    - ...

- Im View Controller:
  - Erzeugen einer Outlet Connection "searchBar"
  - Hinzufügen des `UISearchBarDelegate` Protokolls

- Im Storyboard:
  - Setzen des View Controllers als Delegate der "searchBar"

Collection Views: Beispiel

Laden von Flickr Bildern

- Erzeugen eines API-Keys
  - Z.B. beantragen über https://www.flickr.com/services/api/misc.api_keys.html
- Hinzufügen von Flickr Wrapper Klassen `Flickr` und `FlickrPhoto` (Code von Brandon Trebitowski)
  - `Flickr:` Block-basierte API zur Suche und Rückgabe eines Arrays von Flickr-Bildern
  - `FlickrPhoto`: Zur Erzeugung von Objekten, die ein Flickr-Bild repräsentieren (Thumbnail, Image, Metadatenwie die Foto-ID, ...).
- Ersetzen des Flickr API Keys in `Flickr.m`
  - `#define kFlickrAPIKey @"abcdef0123456789abcdef0123456789"`

Collection Views: Beispiel

Erzeugen eines BridgingHeaders für die importierten Flickr Wrapper Klassen

```
//
//  FlickrSearch-Bridging-Header
//
#import "Flickr.h"
#import "FlickrPhoto.h"
```

Collection Views: Beispiel

Einfügen von Properties im View Controller für Datenstrukturen

- `searches`
  - Alle gestellten Suchanfragen
- `searchResults`
  - Abbildung von Suchanfragen auf `Arrays` von gefundenen `FlickrPhoto`-Objekten

Einfügen einer Property für eine `Flickr`-Instanz

```
//  ViewController.swift

import UIKit

class ViewController: UIViewController, UISearchBarDelegate {

    var searches = Array<String>()
    var searchResults = Dictionary<String,Array<FlickrPhoto>>()
    var flickr = Flickr()

    […]
}
```

Implementierung des `UISearchBarDelegate` -Protokolls

```
//  ViewController.swift

import UIKit

class ViewController: UIViewController, UISearchBarDelegate {

    func searchBarSearchButtonClicked(searchBar: UISearchBar) {
        flickr.searchFlickrForTerm(searchBar.text, completionBlock: {
            (searchTerm:String!, results:[AnyObject]!, error:NSError!) in
            if results.count > 0 {
                if (find(self.searches, searchTerm) == nil) {
                    println("Found \(results.count) photos for \"\(searchTerm)\"")
                    self.searches.insert(searchTerm, atIndex:0)
                    self.searchResults[searchTerm] = results as? [FlickrPhoto]
                }
                dispatch_async(dispatch_get_main_queue(), {
                    // TODO: update collection view
                })
            }
            else {
                println("Error searching Flickr: \(error.localizedDescription)")
            }
        })
        searchBar.resignFirstResponder()
    }
}
```

Collection Views: Beispiel

Einbinden der Protokolle

- UICollectionViewDataSource
- UICollectionViewDelegate
- UICollectionViewDelegateFlowLayout

```
//  ViewController.swift

import UIKit

class ViewController: UIViewController, UICollectionViewDelegateFlowLayout,
        UISearchBarDelegate, UICollectionViewDelegate,
        UICollectionViewDataSource {

    […]

}
```

Collection Views: Beispiel

Implementierung von `UICollectionViewDataSource`

```
//  ViewController.swift

// Liefert die Anzahl von Items innerhalb einer Sektion
func collectionView(collectionView: UICollectionView,
        numberOfItemsInSection section: Int) -> Int {
    let searchTerm = searches[section]
    return searchResults[searchTerm]!.count
}

// Liefert die Anzahl von Sektionen
func numberOfSectionsInCollectionView(collectionView: UICollectionView) -> Int {
    return searches.count
}

// Liefert eine wiederverwendbare bzw. neue Zelle
func collectionView(collectionView: UICollectionView,
        cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCellWithReuseIdentifier(
        "FlickrCell", forIndexPath: indexPath) as UICollectionViewCell
    cell.backgroundColor = UIColor.grayColor()
    return cell
}
```

Collection Views: Beispiel

Implementierung von `UICollectionViewDelegate`

- Wird in diesem Beispiel nicht benötigt...

```
//  ViewController.swift

[...]

func collectionView(collectionView: UICollectionView,
        didSelectItemAtIndexPath indexPath: NSIndexPath) {
    // TODO: Wähle ein Item
}

func collectionView(collectionView: UICollectionView,
        didDeselectItemAtIndexPath indexPath: NSIndexPath) {
    // TODO: Hebe Auswahl des Items auf
}
```

Implementierung von UICollectionViewDelegateFlowLayout

```
// ViewController.swift
[...]

// Bestimmt die Zellgröße
func collectionView(collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        sizeForItemAtIndexPath indexPath: NSIndexPath) -> CGSize {
    // Bestimmen des Fotos für den entsprechenden Index
    let searchTerm = searches[indexPath.section]
    let photo = searchResults[searchTerm]![indexPath.item]
    // Berechnen Zellgröße (falls Foto noch nicht da, setze Größe auf 100x100)
    var retval = photo.thumbnail.size.width > 0 ?
        photo.thumbnail.size : CGSizeMake(100, 100)
    // Fügt dem Foto einen Rand hinzu
    retval.height += 35
    retval.width += 35
    return retval
}

func collectionView(collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        insetForSectionAtIndex section: Int) -> UIEdgeInsets {
    return UIEdgeInsetsMake(50, 10, 50, 10)
}
```

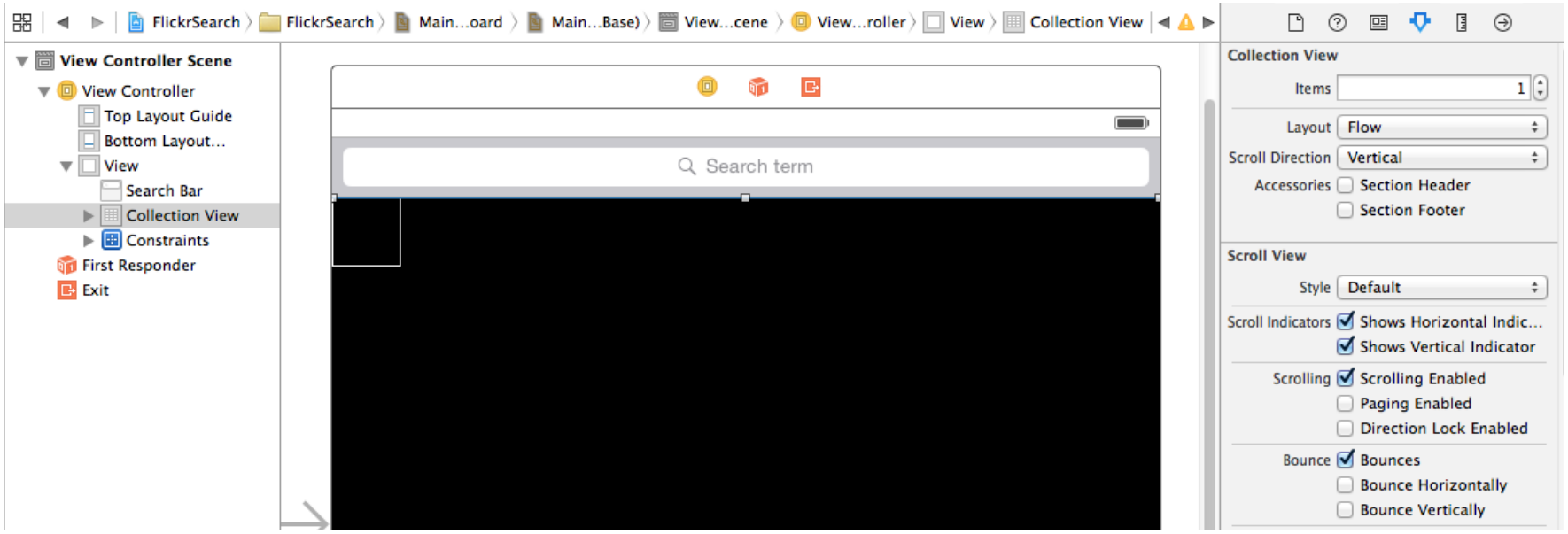
Collection Views: Beispiel

Im Storyboard:

- Hinzufügen einer `UICollectionView`-Instanz
- Anpassen von Attributen
  - Items = 1 (Attribute Inspector): Bewirkt das Hinzufügen einer `UICollectionViewCell`-Instanz

Im View Controller

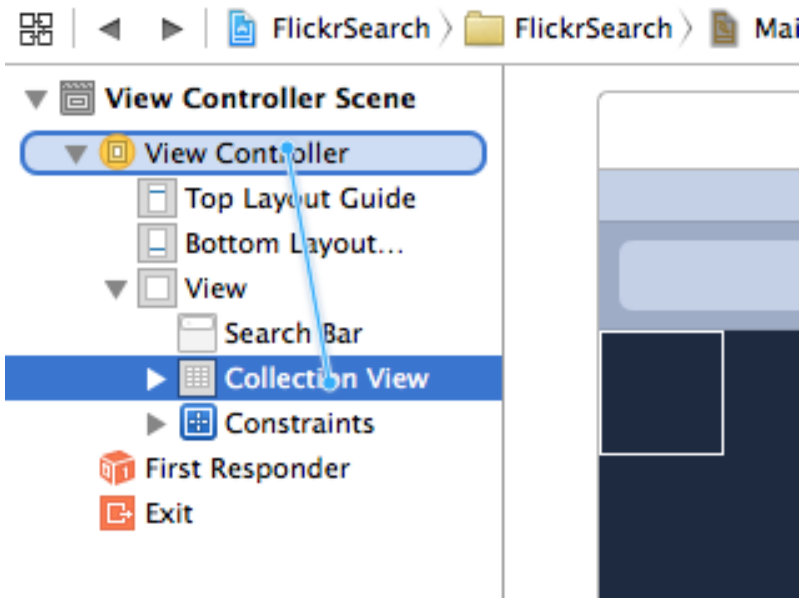
- Erzeugen einer Outlet Connection "collectionView"

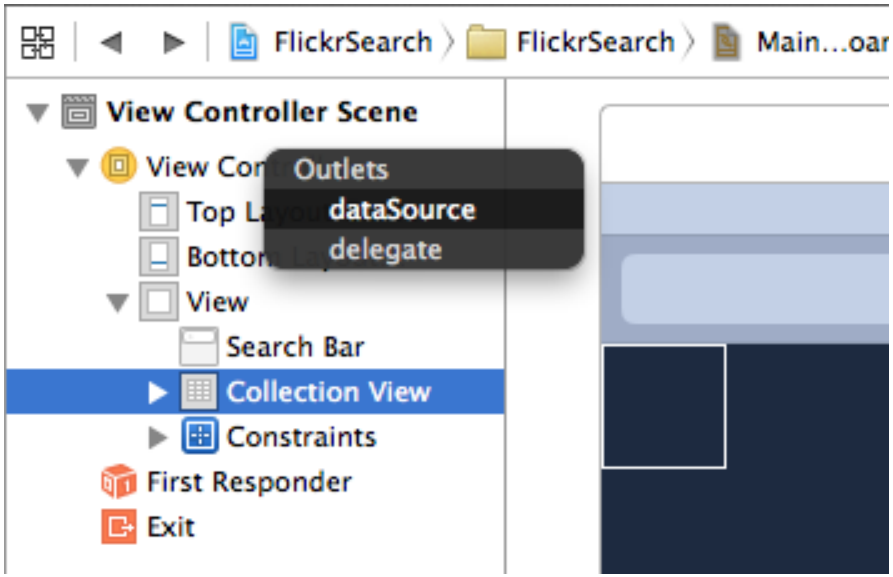


Collection Views: Beispiel

Im Storyboard

- Setzen des View Controllers als Delegate und Data Sources der Collection View





Collection Views: Beispiel

Registrieren von `UICollectionViewCell` als Klasse der zu Erzeugenden Instanzen (testweise)

```
// ViewController.swift

[...]

override func viewDidLoad() {
    collectionView.registerClass(UICollectionViewCell.self,
        forCellWithReuseIdentifier: "FlickrCell")
}
```

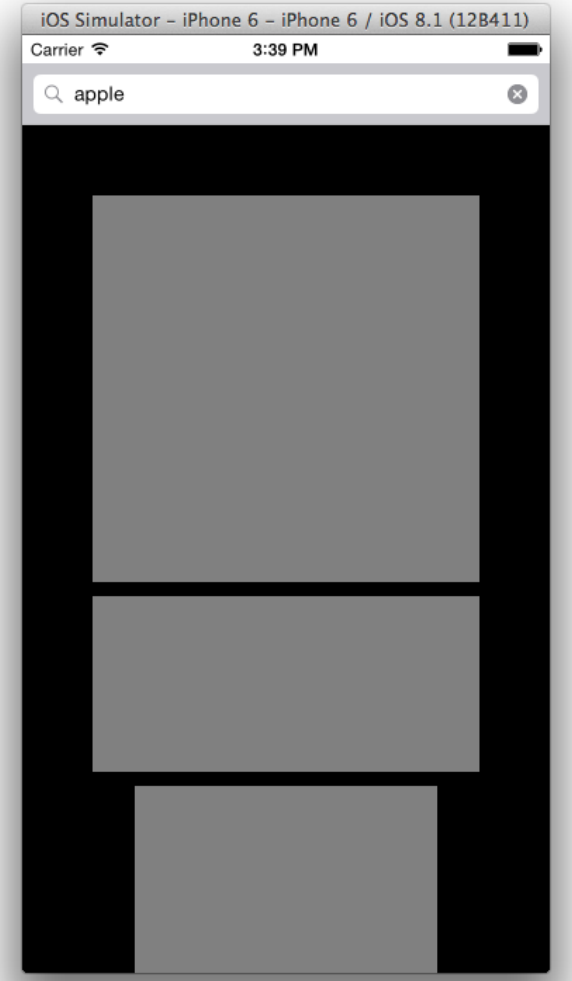
Collection Views: Beispiel

Neuladen der Daten bei Veränderungen in der Data Source

```
//  ViewController.swift

import UIKit

class ViewController: UIViewController, UISearchBarDelegate {

    func searchBarSearchButtonClicked(searchBar: UISearchBar) {
        flickr.searchFlickrForTerm(searchBar.text, completionBlock: {
            (searchTerm:String!, results:[AnyObject]!, error:NSError!) in
            if results.count > 0 {
                if (find(self.searches, searchTerm) == nil) {
                    println("Found \(results.count) photos for \"\(searchTerm)\"")
                    self.searches.insert(searchTerm, atIndex:0)
                    self.searchResults[searchTerm] = results as? [FlickrPhoto]
                }
                dispatch_async(dispatch_get_main_queue(), {
                    self.collectionView.reloadData()
                })
            }
            else {
                println("Error searching Flickr: \(error.localizedDescription)")
            }
        })
        searchBar.resignFirstResponder()
    }
}
```

Collection Views: Beispiel

Ausführen liefert noch keine Bilder



Problem?

- Custom `UICollectionViewCells` noch nicht implementiert...

Collection Views: Beispiel

Erzeugen einer Custom `UICollectionViewCell` Klasse

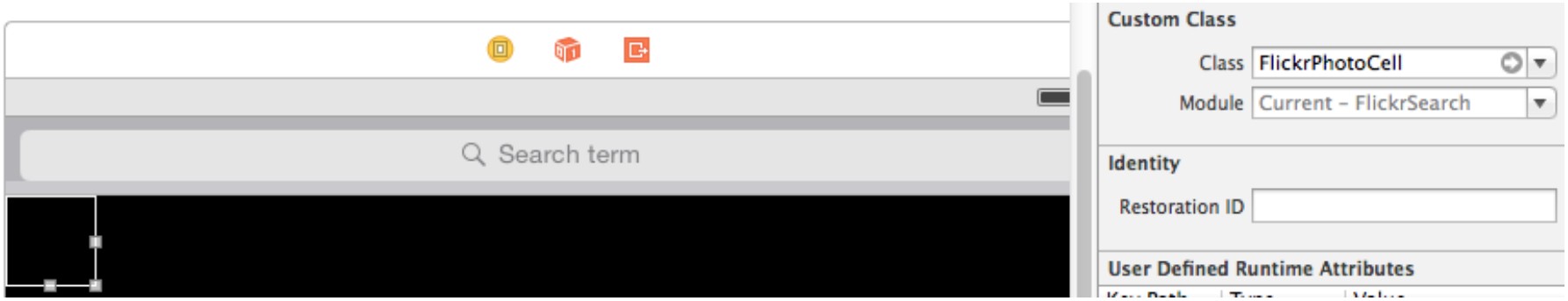
- Achtung: In `ViewController.swift` die Methode `viewDidLoad` jetzt wieder auskommentieren/löschen!

- Anpassen von `FlickrPhotoCell.swift`

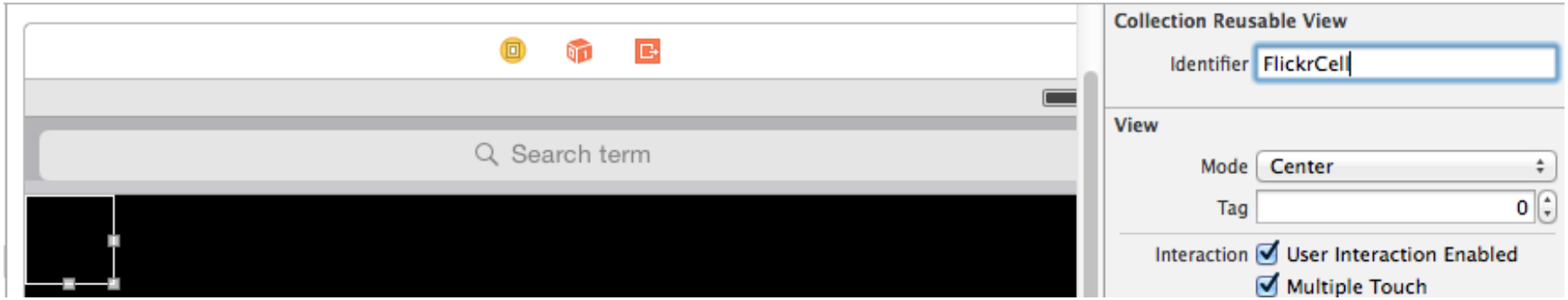
```
//  FlickrPhotoCell.swift

import UIKit

class FlickrPhotoCell: UICollectionViewCell {
    var photo:FlickrPhoto!
}
```

Collection Views: Beispiel

Setzen der Custom Class im Identity Inspector der Collection View Cell

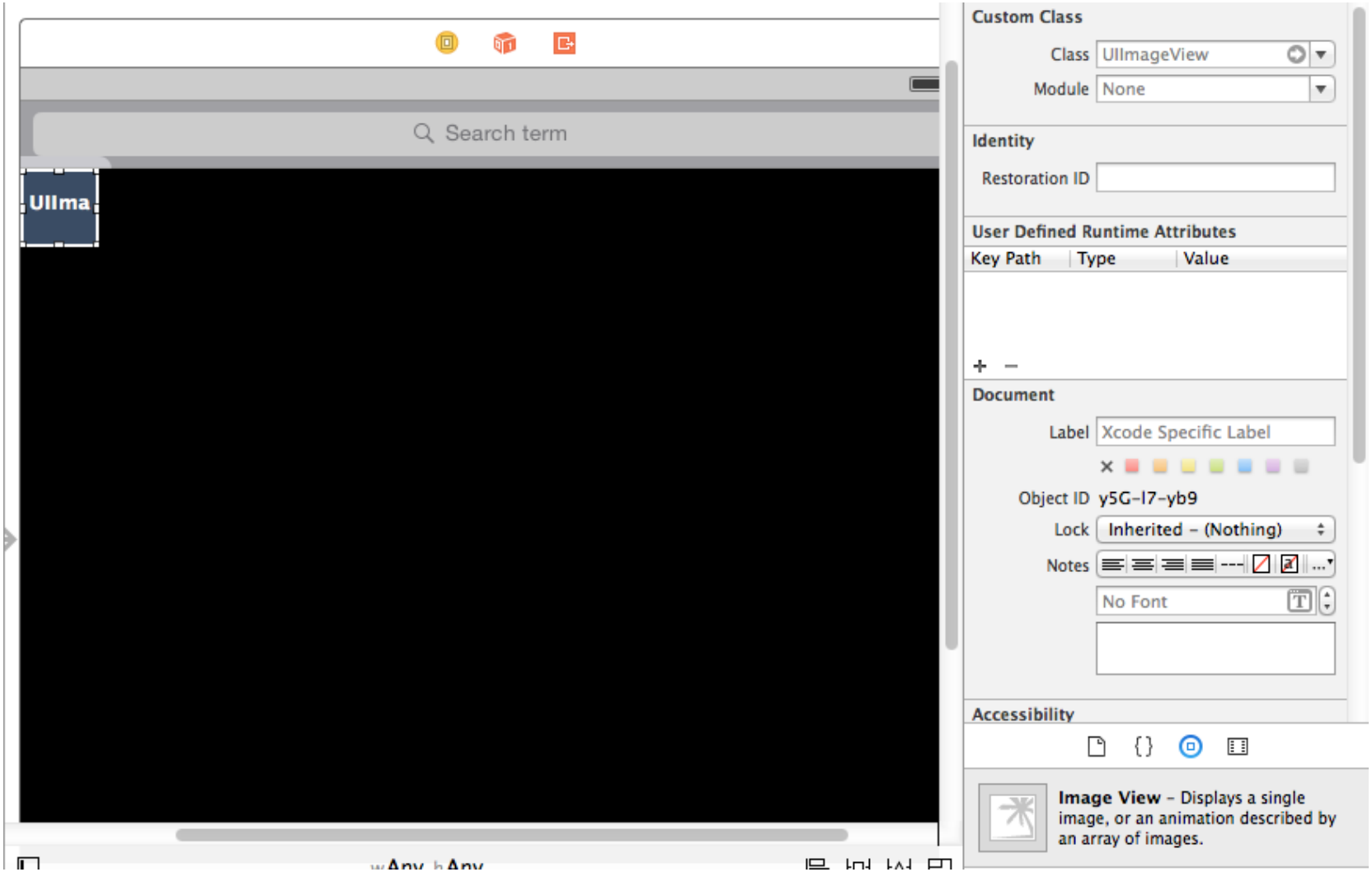


Setzen des Reuse Identifiers im Attribute Inspector auf "FlickrCell"



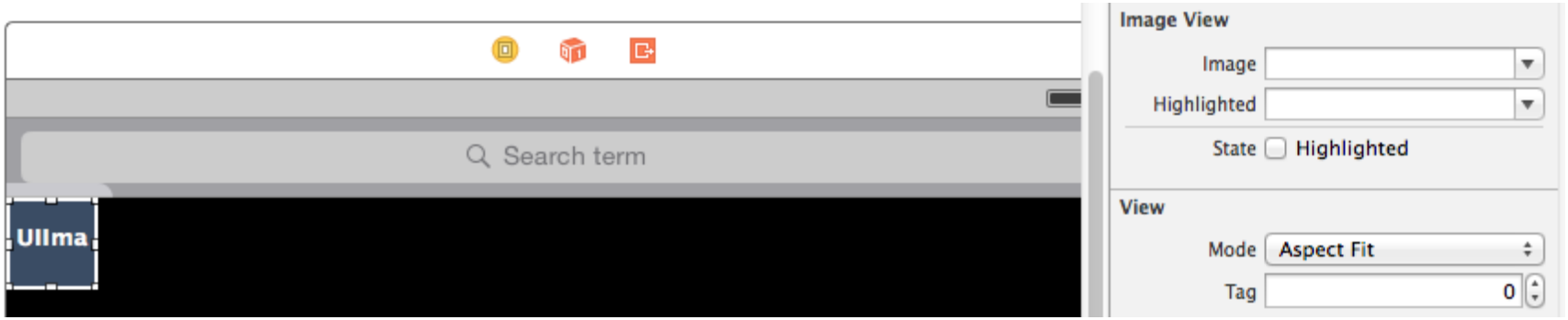
Collection Views: Beispiel

Hinzufügen einer `UIImageView` zur `UICollectionViewCell`

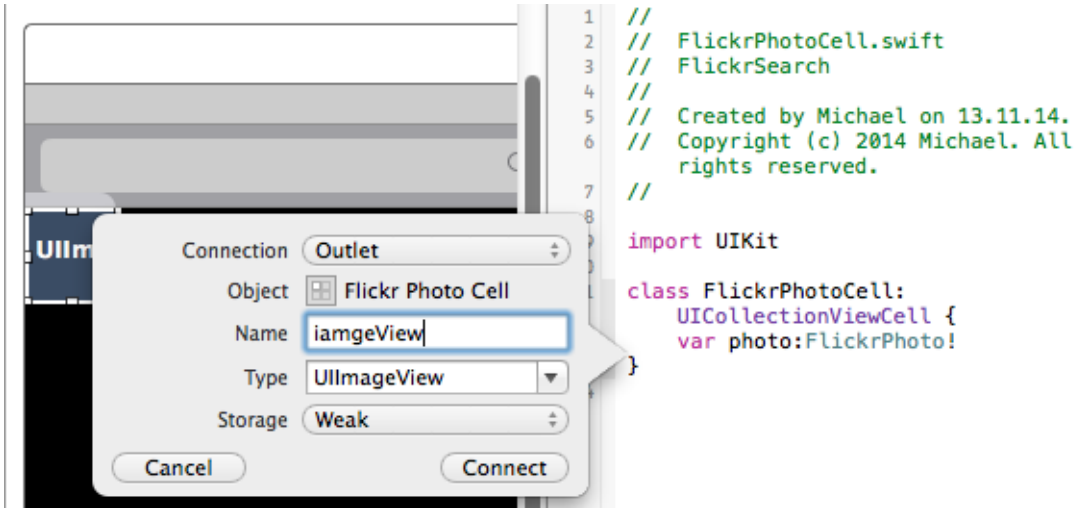


Collection Views: Beispiel

Setzen des View Modes auf "Aspect Fit"



Erzeugen einer Outlet Connection "imageView" zur `UIImageView`



Collection Views: Beispiel

Abändern der Methode `collectionView:cellForItemAtIndexPath:`

```
// ViewController.swift

[...]

func collectionView(collectionView: UICollectionView,
        cellForItemAtIndexPath indexPath: NSIndexPath) ->
        UICollectionViewCell {
    let cell = collectionView.dequeueReusableCellWithReuseIdentifier(
        "FlickrCell", forIndexPath: indexPath) as FlickrPhotoCell
    let searchTerm = searches[indexPath.section]
    cell.photo = searchResults[searchTerm]![indexPath.item]
    return cell
}

[...]
```

Collection Views: Beispiel

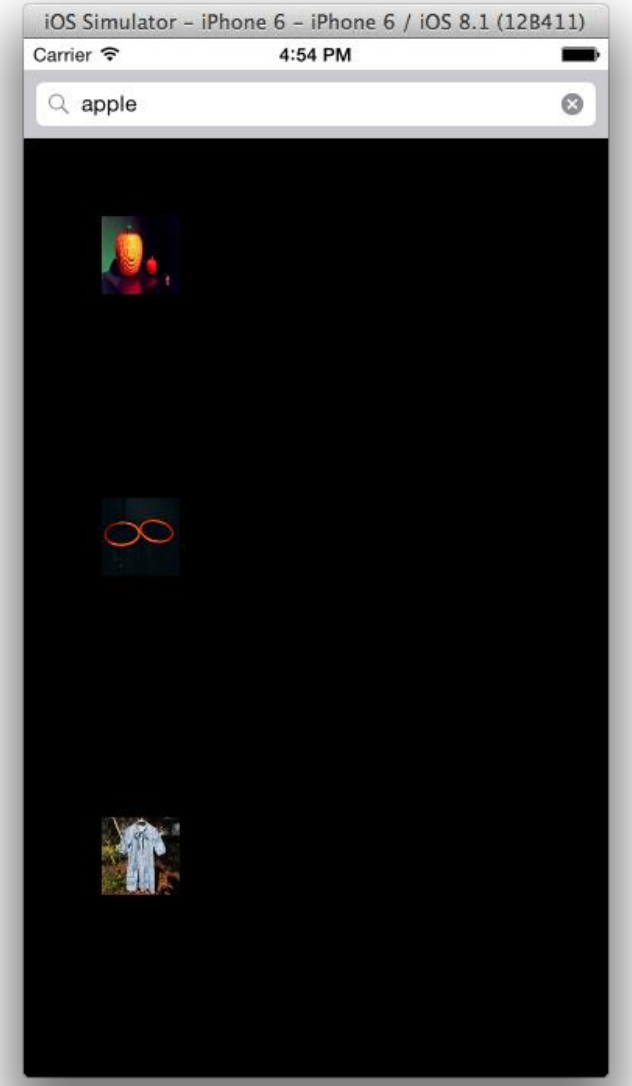
Implementieren der Observer-Methode `didSet` der Property `photo` in der Klasse `FlickrPhotoCell`

```
// FlickrPhotoCell.swift

import UIKit

class FlickrPhotoCell: UICollectionViewCell {

    var photo:FlickrPhoto! {
        didSet {
            imageView.image = photo.thumbnail
        }
    }
    @IBOutlet weak var imageView: UIImageView!

}
```

Collection Views: Beispiel

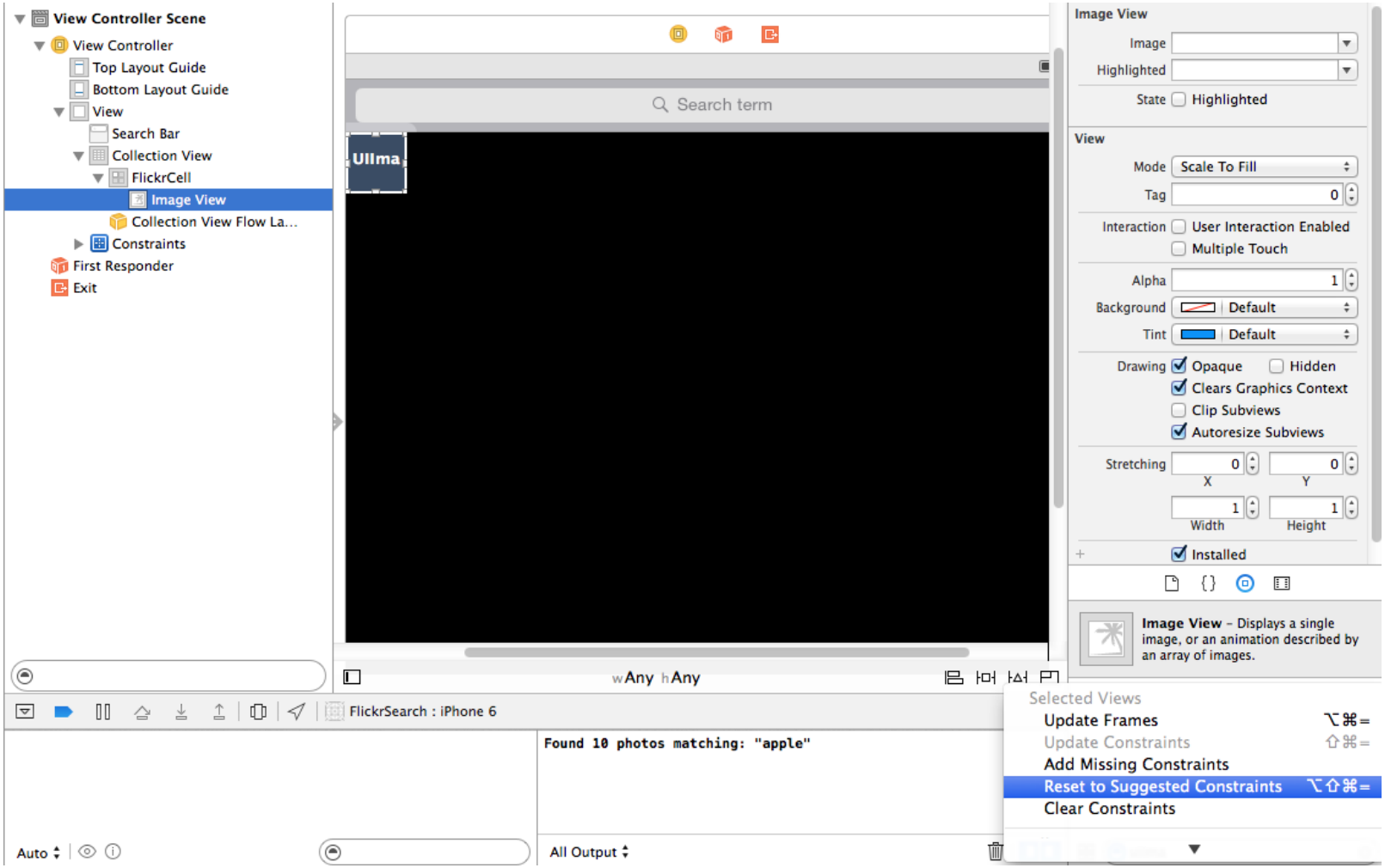
Problem:

- Zellen werden nicht in der korrekten Größe dargestellt



Collection Views: Beispiel

Korrigieren der Layout-Constraints



Collection Views: Beispiel

Ergebnis

Collection Views: Beispiel

Erzwingen mehrerer Spalten durch Anpassen von

```
func collectionView(collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        sizeForItemAtIndexPath indexPath: NSIndexPath) -> CGSize
```

Beispiel

```
// ViewController.swift

// Bestimmt die Zellgröße
func collectionView(collectionView: UICollectionView,
        layout collectionViewLayout: UICollectionViewLayout,
        sizeForItemAtIndexPath indexPath: NSIndexPath) -> CGSize {
    // Berechnen Zellgröße
    let cellWidth = self.collectionView.frame.size.width / 3 - 20
    return CGSizeMake(cellWidth, cellWidth)
}
```

Collection Views: Beispiel

Ergebnis