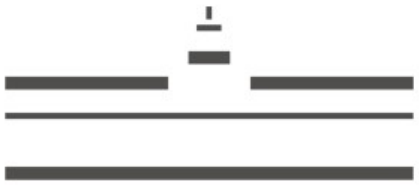


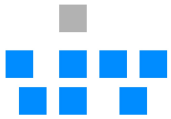
Westfälische
Wilhelms-Universität
Münster

Programmierung für mobile Endgeräte

Cocoa Touch: Die erste App

wissen.leben
WWU Münster

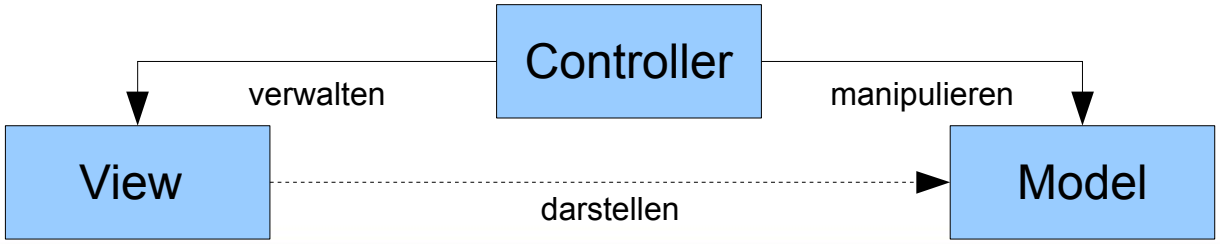
Name: Patrick Förster



ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

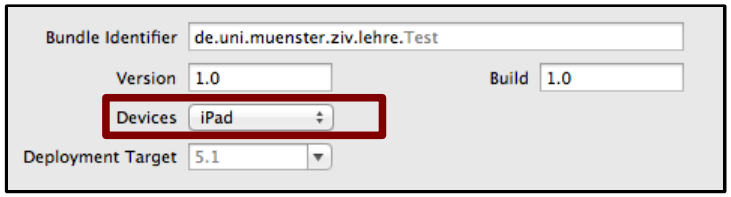
Cocoa: Model View Controller

- Jedes Programm dient am Ende des Tages letztendlich nur zur Darstellung und Manipulation von bestimmten Daten
- Im Idealfall sind die Daten und ihre Darstellung lose gekoppelt, d.h. ein Datum sollte niemals selbst darüber entschieden, wie es dargestellt oder verwaltet wird
- Aus dieser Idee ist das Model-View-Controller (MVC) Pattern entstanden, das Basis aller Cocoa-Applikationen ist
  - Das Model repräsentiert die Daten und sollte unabhängig vom View oder dem Controller sein
  - Der View stellt die Daten da und nimmt Nutzereingabe/-interaktion entgegen
  - Der Controller verwaltet die verschiedenen Views, verarbeittet die Nutzereingaben/-interaktion und manipuliert die Daten



Xcode: Erste App

- Als erste Applikation wird eine iPad-App mit einer einzigen Ansicht erstellt
- Dazu als Templatevorlage „Single View Application“ auswählen
- Als erstes wird in den Einstellungen für das Zielgerät iPad definiert:



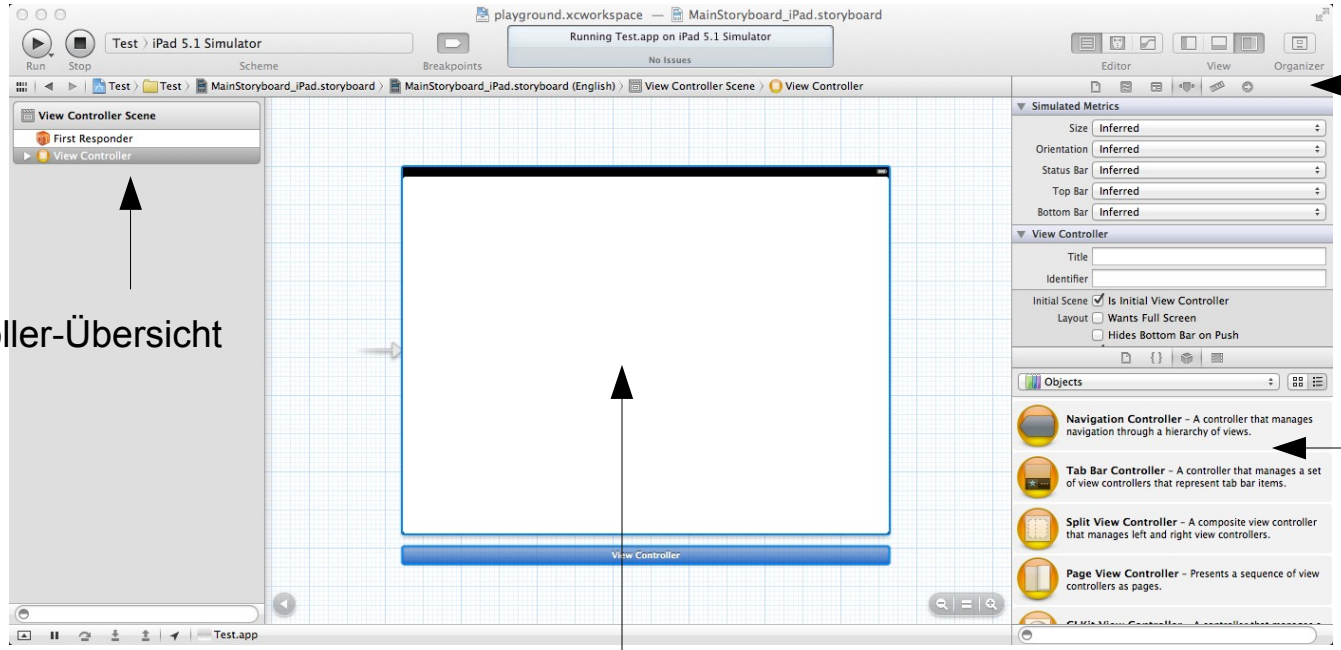
- Die Applikation lässt sich bereits auf dem Simulator starten, man wird allerdings lediglich einen weißen Bildschirm zu Gesicht bekommen
- Durch das Template wurden zwei Klassen erstellt
  - AppDelegate: Die Basis eines jeden iOS-Programmes ist ein Objekt, welches das Protokoll UIApplicationDelegate erfüllt und zur Laufzeit vom Betriebssystem über den Status der Applikation informiert wird
  - ViewController: Da alle Applikationen dem MVC-Pattern gehorchen, wurde als Basis bereits ein Controller erzeugt

Xcode: Erste App

- Wenn man sich die Implementierung der beiden Klassen ansieht, fällt auf, dass diese auf den ersten Blick überhaupt nicht mit einander verbunden sind
- Tatsächlich ist für die ganze Magie das sogenannte Storyboard verantwortlich
- Per Default werden zwei Storyboards – einmal für das iPad einmal für das iPhone - erzeugt
- Storyboards werden ab iOS 5 und Xcode 4.0 unterstützt und sind gewissermaßen eine Weiterentwicklung des Interface-Builders, mit dem zuvor in XCode per Drag-n-Drop Nutzeroberflächen erstellt werden können
- Storyboards sind kein Muss: Viele der nachfolgend gezeigten Vorgehensweise sind analog auch auf den Interface-Builder zu übertragen
- Storyboards bieten im Gegensatz zum Interface-Builder den Vorteil visuell die Navigation durch verschiedene Ansichten designen und nachvollziehen zu können
- Anhand des Storyboards werden zur Laufzeit die Ansichten in der Applikation generiert

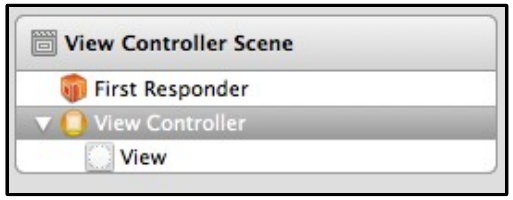
Xcode: Erste App

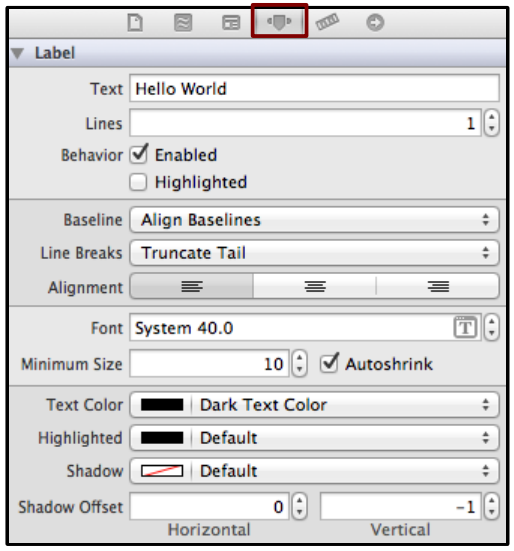
- Die erste App sollte mehr anzeigen als lediglich einen weißen Bildschirm
- Natürlich beginnt man mit „Hello World“
- Durch einen Einfachklick auf das iPad-Storyboard wird dieses geöffnet:



Xcode: Erste App

- In der Übersicht links, klappt man den Controller aus
- Der Controller verfügt durch seine Erzeugung automatisch über ein Attribut namens „View“, welches den Ansichtsbereich der Applikation definiert
- Per Doppelklick zoomt das Storyboard in den View und man hat die Möglichkeit neue Objekte zur Ansicht hinzuzufügen (dies geht in der Tat nur in der hereingezoomten Ansicht!)
- Unten rechts befindet sich der Objekt-Navigator, hier kann man nun ein Label-Objekt per Drag-n-Drop im Darstellungsbereich platzieren
- Über den „Attributt-Inspector“ rechts kann außerdem das Aussehen und der Text für das Label definiert werden





Xcode: Outlets und Actions

- Beim Umgang mit dem Interface-Builder (Storyboard) wird die Trennung zwischen View und Controller sehr deutlich:
  - Das soeben hinzugefügte Label taucht nirgendwo im Code
  - Auch das im Storyboard angezeigte View-Objekt ist nicht zu finden
  - Zur Laufzeit werden diese Objekte durch die über das Storyboard angegebene Beschreibung erzeugt und dem Controller, bzw. dem View hinzugefügt
- Es besteht die Möglichkeit einzelne Elemente durch ein „Tag“ zu identifizieren
- Dazu muss im Attribute-Inspector unter „View“ im Feld „Tag“ eine Nummer vergeben werden
- Im Controller kann über den View auf das Objekt zugegriffen werden
- Allerdings ist ein solcher Zugriff sehr fehleranfällig:

```
id temp = [self.view viewWithTag: TAG];
```

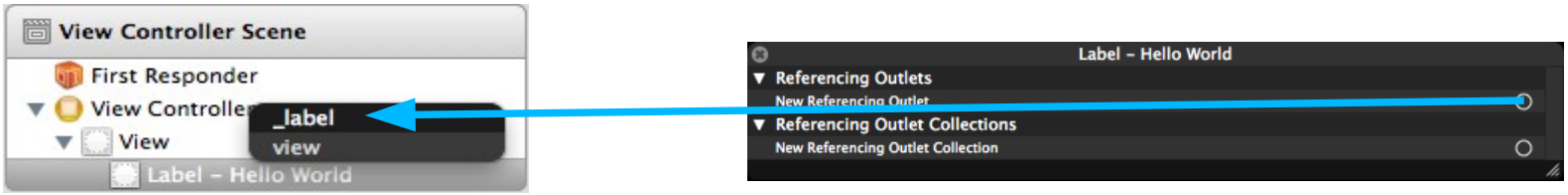
  - Ein Tag ist immer vom Typ NSInteger und damit solange kein entsprechender Enum deklariert wurde, nicht sehr aussagekräftig
  - Ein Tag muss nicht eindeutig sein, d.h. mehrere Objekte können dem selben Tag zu geordnet werden
  - Die Rückgabe des Aufrufs ist unspezifiziert

Xcode: Outlets und Actions

- Eine eindeutige Beziehung zwischen einem im Interface-Builder hinzugefügten Objekt und einer Referenz in einem Controller wird über sogenannte Outlets erstellt
- Als Referenz legt man im Controller ein Zeigerattribut des benötigten Typs an und markiert diese mit dem Schlüsselwort IBOutlet:

```
@interface PGViewController : UIViewController {
    IBOutlet UILabel* _label;
}
```

- Im Storyboard öffnet man per Rechtsklick auf das Label-Objekt das „Connection“-Menü und verbindet per Drag-n-Drop den Punkt neben „New Referencing Outlet“ mit dem Controller, woraufhin sich ein Kontext-Menü mit den möglichen Outlets öffnet. Hier wählt man das _label.



Xcode: Outlets und Actions

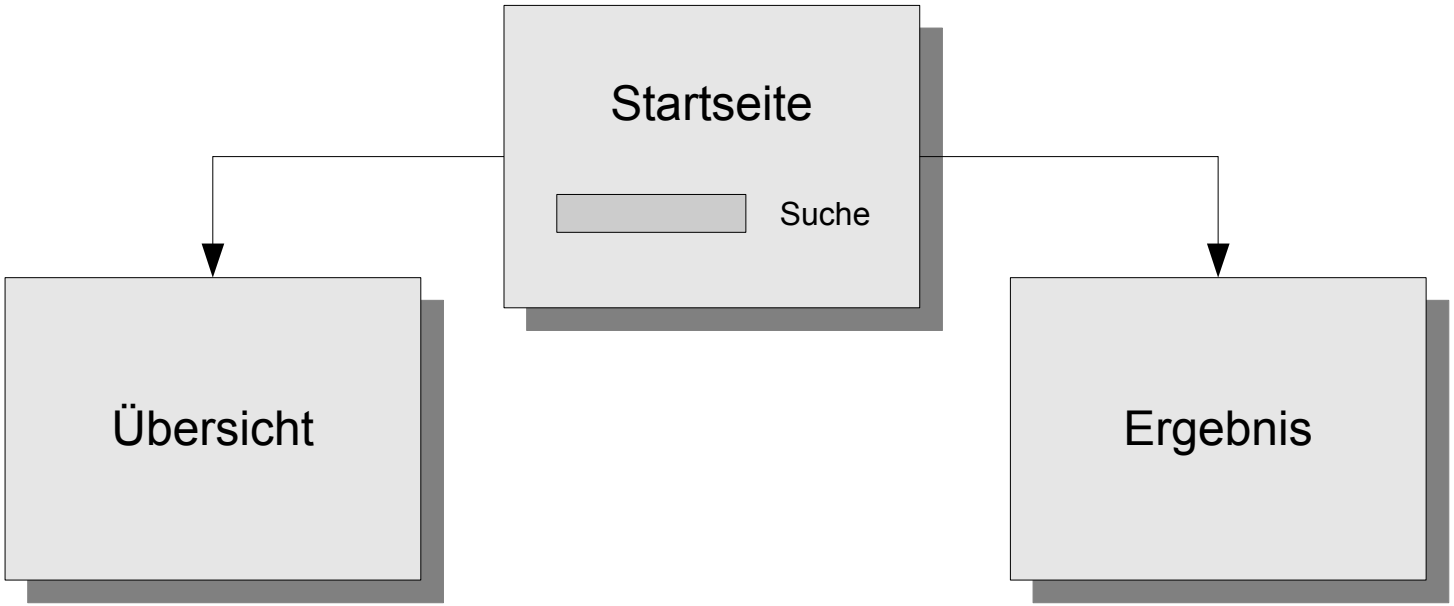
- Sobald der View für den aktuellen Controller erzeugt wurde, werden automatisch die Referenzen im Controller mit Zeigern auf die entsprechenden Objekte gefüllt.
- Ganz ähnlich wird mit den IBActions verfahren, die eine bestimmte Nutzerinteraktion mit einer bestimmten Callback-Methode im Controller verbinden
- Zu Demonstration fügt man im Storyboard einen neuen Button („Round Rect Button“) hinzu. Das zugehörigen Connections-Menü zeigt eine Reihe von möglichen Events an
- Eine Callback-Methode wird durch IBAction als Rückgabetyp markiert und erwartet immer einen Parameter: den Sender des Events

```
-(IBAction)ontouch:(id)sender {
    _label.text = [NSString stringWithFormat: @"Hello %@", sender];
}
```

- Nun kann im Storyboard analog zum Outlet z.B. der „Touch Down“-Event des Buttons mit der neuen IBAction verbunden werden

Die Kennzeichen-Applikation

- Die Applikation soll die Landkreis/Stadt Abkürzung auf einem Kennzeichen auflösen
- Zudem soll sie eine Übersicht über alle Abkürzungen anbieten
- Für ein wenig Abwechslung werden die Informationen auf drei Ansichten verteilt:

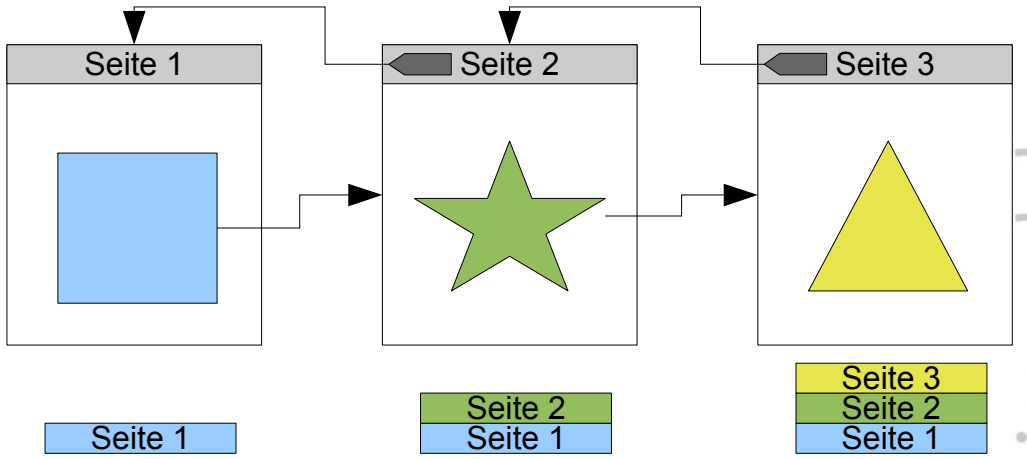


Wahl des ViewControllers

- Die „Hello World“ Applikation basierte auf einer einzigen Ansicht, auf der alle Informationen angezeigt wurden
- Will man die Informationen auf mehrere Views verteilen, benötigt es einen Controller, der dieses unterstützt
- Das Cocoa-Touch-Framework liefert zu diesem Zweck gleich eine Reihe von möglichen ViewControllern
  - *NavigationController*: Navigation durch hierarchisch angeordnete Ansichten
  - *TabBarController*: Eine Reihe von Ansichten, die über eine Tab-Leiste gesteuert werden
  - *SplitViewController*: Der Bildschirm wird in zwei Ansichten aufgeteilt
  - *PageViewController*: Eine Reihe von Ansichten, die analog zu Buchseiten aneinandergereiht sind
- Innerhalb dieser Controller, können natürlich wieder andere geschachtelt werden
- Man ist ebenso nicht auf die vordefiniert Controller beschränkt
- Allerdings: Die meisten Nutzer akzeptieren eher das „Vertraute“ als das „Neue“

Wahl des ViewControllers (II)

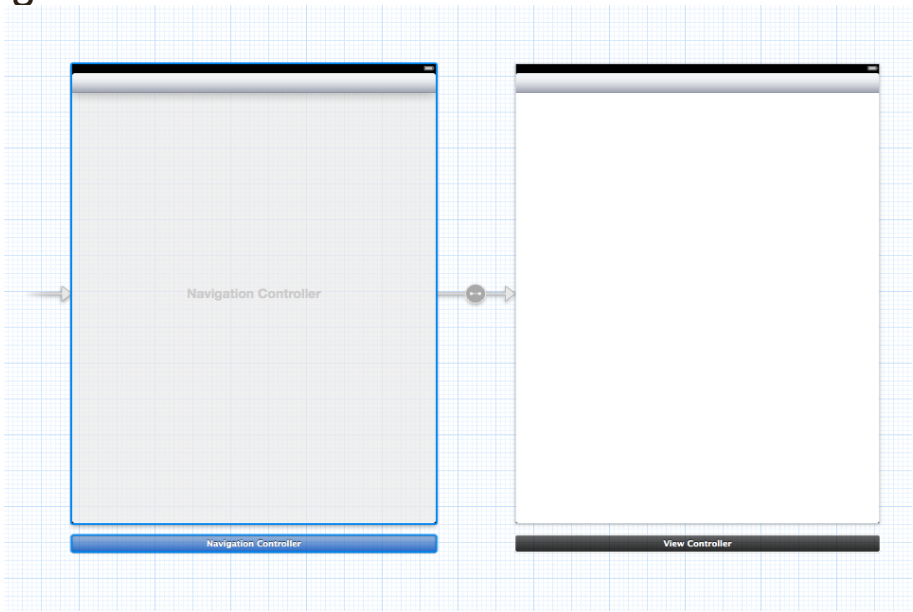
- Für die Kennzeichen-App wird der NavigationController eingesetzt:
  - Die durch den Controller verwaltetet Ansichten werden als Stack angeordnet, d.h. der Nutzer „sieht“ immer nur die oberste Ansicht
  - Jedes Mal wenn zu einer neuen Ansicht navigiert wird, wird diese auf den Stack gelegt, so dass von der aktuellen Ansicht immer zurück zur vorherigen navigiert werden kann
  - Diese Navigation wird standardmäßig immer in der Navigationsleiste am oberen Bildschirmrand abgebildet (Generiert per Storyboard, ist dies „geschenkt“)



  - Der Wechsel von einer zur nächsten Ansicht wird als „Übergang“ (Segue) bezeichnet und lässt sich ebenfalls einfach über das Storyboard definieren

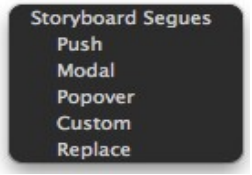
NavigationController

- Zur Erzeugung eines neuen Projektes basierend auf einem NavigationController existiert kein Template
- Man erstellt zu erst eine „Single View Application“
- Im Storyboard markiert man den autogeneriertne Default-Controller und wählt dann unter Editor „Embed In“ → „Navigation Controller“
- Ein NavigationController wird erzeugt und der bereits vorhandene Controller als erster Controller in der Hierarchie markiert („Relationship“ Verbindung zwischen den beiden Controllern)
- Löscht man die Beziehung würde man lediglich einen schwarzen Bildschirm mit Navigationsleiste sehen



Storyboard-Übergange (Segues) (I)

- Um eine neue Ansicht auf den Stack schieben zu können, muss ein neuer Controller her, der diese Ansicht verwaltet
- Für die Ergebnisansicht in der Kennzeichen-App genügt es im Storyboard einen einfachen ViewController hinzuzufügen
- Damit der View gewechselt werden kann, benötigt es nun noch einem Objekt, das eine Nutzeraktion entgegen nehmen kann. Dazu fügt man z.B. einen Button in den View des ersten Controllers ein
- Ein Übergang wird nun wieder per Drag-n-Drop angelegt, in dem man die Maus mit gedrückter Ctrl-Taste vom Button auf den gewünschten nächsten View zieht
- Sobald man die Maustaste loslässt, bekommt man ein Kontextmenü angezeigt



- Für den Übergang des NavigationControllers wählt man „Push“
- Der zweite Controller wird nun automatisch vom NavigationController verwaltet erkannt und bekommt eine Navigationsleiste spendiert
- Zur besseren Übersicht sollte man die Leiste im Attribute-Inspector benennen

Storyboard-Übergange (II)

- Aktuell ist der zweite ViewController noch keiner Klasse zugewiesen, so dass per Default eine Instanz der Klasse Basisklasse UIViewController erzeugt wird
- So können für diesen Controller allerdings keine IBOutlets oder IBActions definiert werden
- Um dies zu ändern, erzeugt man eine neue Klasse, die von UIViewController erbt und weißt diese dem Controller über den Identity-Inspector zu 
- Als nächstes wird zum View des ersten Controllers ein Textfeld hinzugefügt, der in der App eine Art „Suchschlitz“ darstellen soll und im Ergebnis-Controller ein Label, das die Suchanfrage erneut ausgibt
- Die UI-Elemente dann per IBOutlet mit dem jeweiligen Controller verknüpfen
- Damit nun die Daten aus dem ersten View an den Controller für den zweiten übertragen werden können, muss zum ersten Mal programmiert werden
- Bevor ein Übergang eingeleitet wird, wird der aktuelle Controller über folgende Methode darüber informiert, dass ein Übergang bevorsteht:

```
-(void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
```

Storyboard-Übergange (III)

- Das Attribute `segue` verweist dabei auf den bevorstehenden Übergang und bietet Zugriff auf den Ziel-Controller, während `sender` auf das UI-Objekt verweist, welches den Übergang eingeleitet hat
- Naiv könnte man nun das Label des zweiten Controllers per Property zugreifbar machen und innerhalb der `segue`-Methode setzen, allerdings wurde das Storyboard für den Controller zu diesem Zeitpunkt noch nicht umgesetzt, d.h. das Label-Objekt ist noch `nil`.
- Eine mögliche Lösung wäre zum Zwischenspeichern der Suchanfrage im zweiten Controller ein neues Property-Attribut hinzuzufügen und dieses dann in der `segue`-Methode zu setzen.

```
-(void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    PGResultViewController* ctrl = (PGResultViewController*) segue.destinationViewController;
    ctrl.searchQuery = _textField.text;
}
```

- Da die `segue`-Methode für jeden Übergang aufgerufen wird und es durchaus mehrere pro Controller geben kann, sollte man die einzelnen Übergänge benennen

Storyboard-Übergange (IV)

- Dazu wählt man im Storyboard den Übergang aus und setzt im Attribute-Inspector einen Bezeichner (Identifier)
- In der `segue`-Methode kann man den Bezeichner dann über das `segue`-Objekt abfragen und anhand des Resultats die gewünschte Übergangslogik durchführen:

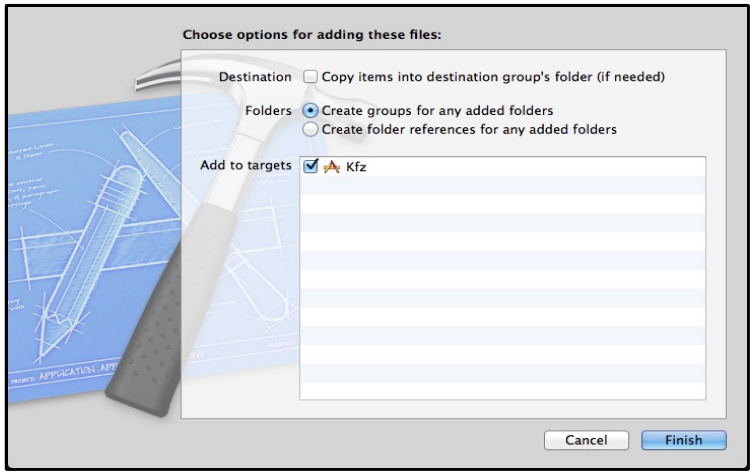
```
-(void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    if ([segue.identifier isEqualToString:@"toResult"]) {
        ...
    }
}
```

- Sobald ein Controller erzeugt und anhand des Storyboards „gefüllt“ wurde, wird die Methode viewDidLoad aufgerufen. Hier kann das Label nun gesetzt werden.

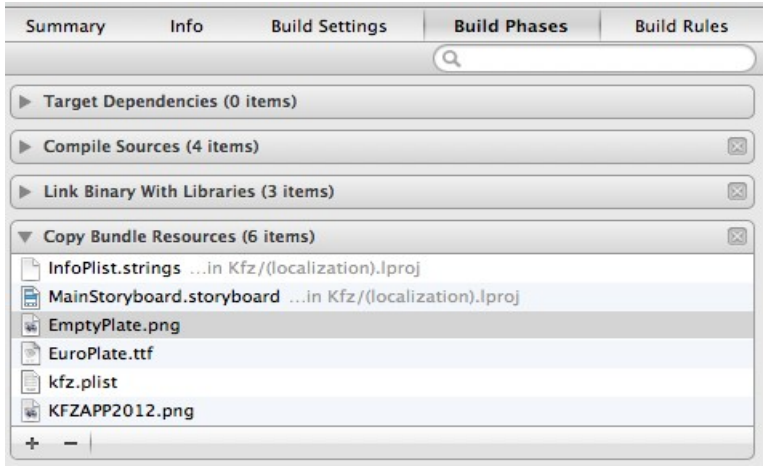
```
- (void)viewDidLoad {
    [super viewDidLoad];
    _searchLabel.text = _searchQuery;
}
```

Xcode: Ressourcen

- Ressourcen sind Grafiken, Fonts, XML-Dateien, ect., die von der App genutzt werden und nicht direkt im Source-Code definiert werden können
- Neue Ressourcen können einfach per Drag-n-Drop zu Xcode hinzugefügt werden
- Ressourcen wie Bilder, die in der App benötigt werden, müssen mit der App ausgeliefert werden und sollten in den Einstellungen unter „Build-Phases“ → „Copy Bundle Ressources“ auftauchen



Drag-n-Drop Dialog



Build Settings

Ressource (Bilder)

- Dem Projekt werden vier Ressourcen hinzugefügt:
  - *EmptyPlate.png*, *KFZAPP2012.png*: Grafiken, die in der App zur „Aufhübschung“ genutzt werden
  - *EuroPlate.ttf*: Font zur Darstellung von Kennzeichen
  - *kfz.plist*: XML-basierte Auflistung aller Landkreis/Stadt → Kennzeichen Paare
- Zur erst wird das Logo (*KFZAPP2012.png*) in den ersten View integriert. Dazu fügt man ein ImageView-Objekt ein und setzt das Image-Attribut über den Inspector
- Im Ergebnis-View wird die leere Plakette (*EmptyPlate.png*) analog hinzugefügt
- Das bereits existierende Label im Ergebnis-View wird nun in den freien Raum der Plakette platziert und die Größe angepasst
- Da die Plakette nach dem Label hinzugefügt wurde, wird sie nun das Label verdecken
- Die Zeichenreihenfolge kann im „Document Outline“ (links) geändert werden (das zuletzt zu zeichnende ist ganz in der Reihenfolge ebenfalls das letzte)

Ressource (Font)

- Als nächstes soll das innerhalb der leeren Plakette platzierte Label auch in der zugehörigen Schriftart dargestellt werden
- Leider wird das Font nicht im Storyboard angeboten, weshalb das Festlegen der Schriftart programmatisch geschehen muss
- Ein geeigneter Zeitpunkt hierfür ist die `viewDidLoad`-Methode, die aufgerufen wird, nachdem der View anhand des Storyboardes geladen wurde

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIFont* font = [UIFont fontWithName:@"Euro Plate" size:100];
    [_searchLabel setFont:font];
    _searchLabel.text = [_searchQuery uppercaseString];
}
```

- Man beachte, dass man den Font-Namen, nicht den Namen der Datei angeben muss (zu finden bspw. über die MacOS „Schriftsammlung“)
- Abschließend muss die Schriftart noch bekannt gemacht werden: Dazu in der Datei ?*-Info.plist einen neuen Schlüssel hinzufügen: „Fonts provided by application“ und als „Item“ den Dateinamen(!) eintragen

Ressource (pList)

- pList-Dateien (Property-List) werden im Mac-Umfeld zur Serialisierung von Objektstrukturen genutzt
- Das Dateiformat ist seit Mac OS 10.0 XML oder seit 10.2 auch JSON
- Wie auf der vorherigen Folien gesehen, werden pListen auch zur Konfiguration der Apps genutzt
- Zudem bietet Xcode eine komfortable Möglichkeit solche Dateien zu verändern ohne in die XML- oder JSON-Syntax einsteigen zu müssen
- Die Datei kfz.plist enthält alle Kennzeichen – Landkreis/Städte Paar und kann nun dazu genutzt werden die Nutzereingabe zu mappen
- Zum Glück bietet die Klasse NSMutableDictionary eine relativ einfach Möglichkeit pList-Dateien in eine Schlüssel-Wert Tabelle (== Dictionary) umzuwandeln

```
NSString* plistPath = [[NSBundle mainBundle] pathForResource:@"kfz" ofType:@"plist"];
NSMutableDictionary* dictonary = [[NSMutableDictionary alloc] initWithContentsOfFile:plistPath];

if (dictonary && [dictonary objectForKey:_searchQuery]) {
    ...
}
```

Die App

- Dieser Code kann nun erneut in der `viewDidLoad`-Methode des Ergebnis-Controllers angebracht werden
- Um das Ergebnis darzustellen wird zudem ein neues Label + Outlet in den Ergebnis-View integriert und ebenfalls in der `viewDidLoad`-Methode gesetzt