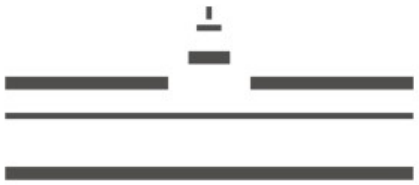


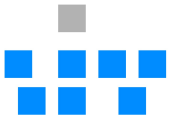
Westfälische
Wilhelms-Universität
Münster

Programmierung für mobile Endgeräte

Core Data (II) und nutzerdefinierte View-Bausteine

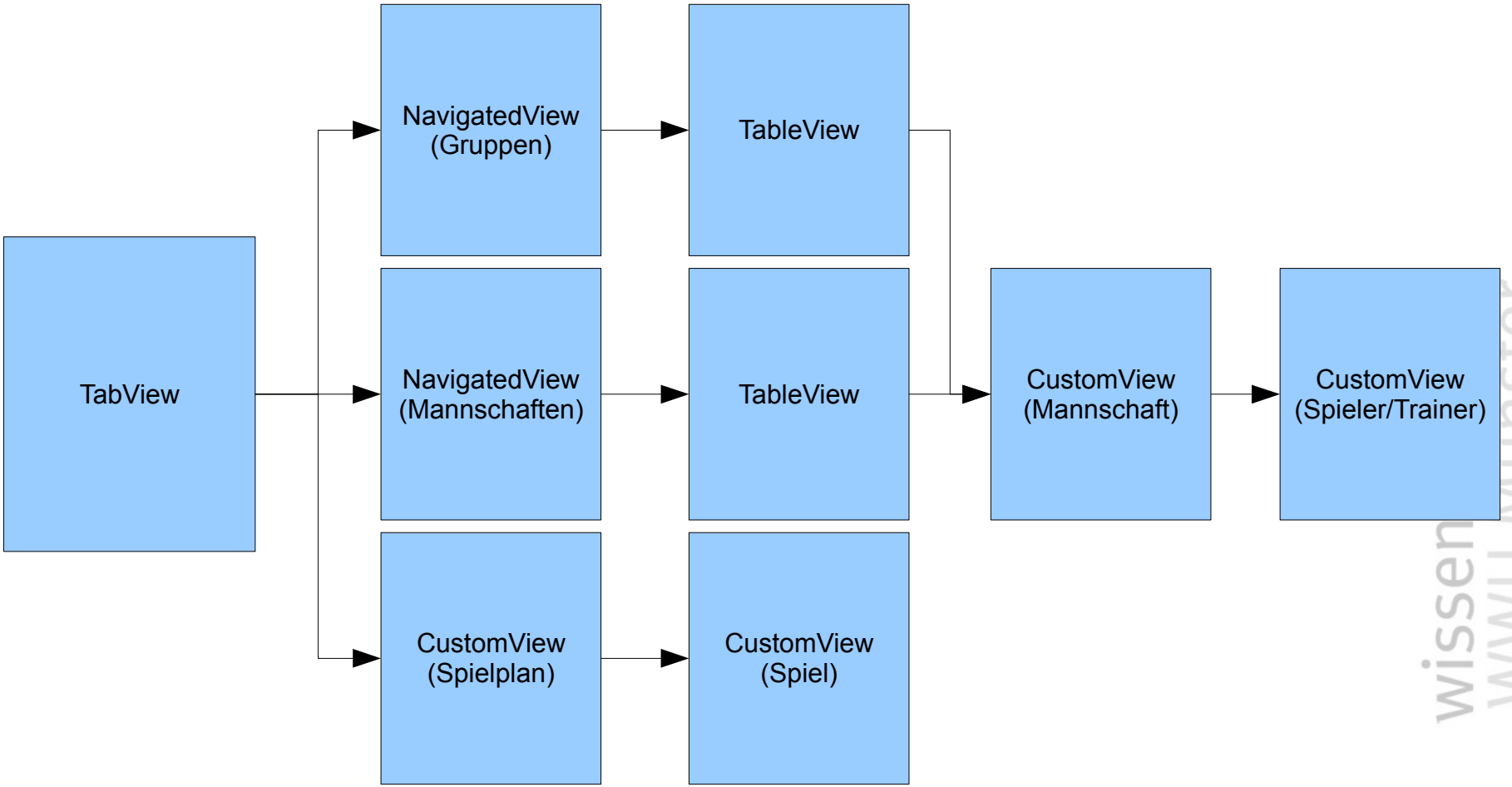
wissen.leben
WWU Münster

Name: Patrick Förster



ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Die EM-App (erster Entwurf)

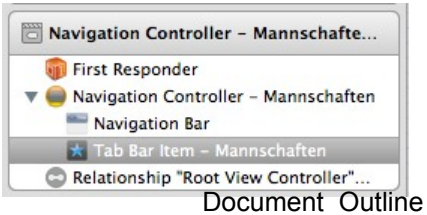


Die EM-App (erster Entwurf)

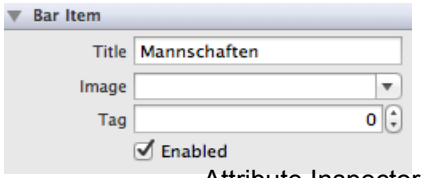
- Als Basis der App ist ein TabViewController vorgesehen, über den in einer Menüleiste am unteren Bildschirmrand drei Unteransichten ausgewählt werden können:
  - Gruppen: Auflistung aller Gruppen samt Mannschaften in einer Tabelle aufgeteilt nach Gruppen
  - Mannschaften: alphabetische Auflistung aller Mannschaften
  - Spielplan: Eine Übersicht über den Spielplan
- Von den beiden Ansichten „Gruppen“ und „Mannschaften“ kann weiter navigiert werden zu einer Mannschaftsansicht, in der Mannschaftsinfos, die Spieler und der Trainer aufgeführt sind
- Von dort gelangt man zu einer Detailansicht für Spieler bzw. Trainer
- Dieses Navigationskonzept legt nahe, dass die Gruppen- und Mannschaftsansicht durch einen NavigationController verwaltet werden
- Der Spielplan wird auf eine komplett selbstdefinierte View-Controller-Struktur zurückgreifen und per „Popover“ Details zu den einzelnen Spielen bieten

Die EM-App (TabBarController)

- Falls noch nicht vorhanden im Storyboard einen neuen TabBarController hinzufügen und über den „Attribute Inspector“ als „Is Initial View Controller“ markieren
- Die beide per Default erzeugten ViewController können gelöscht und durch TableViewController ersetzt und dann per „Editor“ → „Embed in“ → „Navigation Controller“ in einen NavigationController eingebettet werden
- Per *Ctrl + Klick* auf den TabBarController und ziehen auf jeweils einen der beiden NavigationController müssen Beziehungen zwischen den Controllern hergestellt werden
- Die NavigationController erhalten so automatisch einen TabBarItem, der nun per Auswahl im „Document Outline“ markiert und dann im „Attribute Inspector“ benannt werden kann („Gruppen“, bzw. „Mannschaften“)



Document Outline



Attribute Inspector

Die EM-App (TableView: Gruppen)

- Wie gehabt wird nun eine neue TableViewController-Klasse erzeugt und im Storyboard für die Gruppenansicht zugeordnet
- Als nächstes muss das DataSource-Protokol implementiert werden:
- Da es festdefiniert 4 Gruppen á 4 Mannschaften gibt, können die beiden Methoden numberOfSectionsInTableView und tableView:numberOfRowsInSection trivial implementiert werden

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {return 4;}
```

- In der Methode tableView:cellForRowAtIndexPath wird zu nächst eine Zelle erzeugt (evtl. muss der Zellenbezeichner noch im Storyboard über den „Attribute Inspector“ gesetzt werden) und nachfolgend die Zelle mit Daten gefüllt
- Dazu muss herausgefunden werden, welche Mannschaft zum mitgelieferten NSIndexPath gehört
- Dazu wird einfach (wie in der Kennzeichen-App) die Gruppe anhand des Abschnittes identifiziert und dann die Mannschaft mit dem gesuchten row-Index zurückgegeben

Die EM-App (TableView: Gruppen II)

```
NSManagedObjectContext* ctx = [self managedObjectContext];

NSEntityDescription *entityDescription = [NSEntityDescription entityForName:@"Group" inManagedObjectContext:ctx];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:entityDescription];
```

- Der hier erzeugte Fetch-Request repräsentiert die trivialsten aller Anfrage: Bitte gibt mir alle Objekte eines bestimmten Typs
- Dazu muss dem Request lediglich die Beschreibung der gesuchten Entität mitgegeben werden

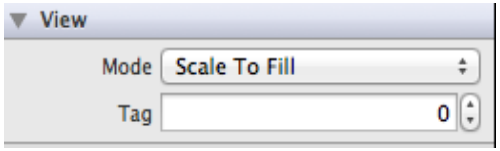
```
NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"name" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];

NSArray* groups = [self fetchArray:request];
return [groups.teams objectAtIndex:index.row];
```

- Damit die Gruppe eindeutig dem Abschnitt zugeteilt werden kann, wird zudem erwünscht, dass die Rückgabe alphabetisch sortiert ist
- Abschließend wird der Request abgesendet (vgl. Folie 19) und aus dem Ergebnis die gesuchte Mannschaft herausgesucht

Die EM-App (TableView: Mannschaften)

- Im Gegensatz zum bekannten Prozedere wird für die Mannschaftsansicht kein neuer Controller erstellt, sondern der zuvor für die Gruppenansicht erstellt mitverwendet
- Dazu sind ein paar Anpassungen erforderlich, insb. muss irgendwie festzustellen sein, für welchen View ein Controller gerade als Delegate fungiert
- Dazu werden die Views der Controller über den „Attribute Inspector“ getagged
- Zur besseren Identifikation im Quellcode wird zudem ein Enum definiert, über dass per Switch-Anweisung nun das DataSource-Protokoll angepasst werden kann
- In der Mannschaftsansicht gibt es nur einen Abschnitt und insgesamt gibt es 16 Mannschaften:




```
typedef enum {
    PG_GROUP_CONTROLLER,
    PG_TEAM_CONTROLLER

} PG_VC_TAG;
```

```
switch (self.view.tag) {
    case PG_GROUP_CONTROLLER: return 4;
    case PG_TEAM_CONTROLLER: return 1;
}
```

```
switch (self.view.tag) {
    case PG_GROUP_CONTROLLER: return 4;
    case PG_TEAM_CONTROLLER: return 16;
}
```

Die EM-App (TableView: Mannschaften II)

- Für die Zellen wird ein einfacher Request gestellt, alle Mannschaften zu liefern und das Resultat alphabetisch sortiert

```
-(NSArray*) fetchTeams {
    NSManagedObjectContext* ctx = [self managedObjectContext];
    NSEntityDescription *entityDescription = [NSEntityDescription entityForName:@"Team" inManagedObjectContext:ctx];
    NSFetchRequest *request = [[NSFetchRequest alloc] init];
    [request setEntity:entityDescription];
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc] initWithKey:@"name" ascending:YES];
    [request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];
    return [self fetchArray:request];
}
```

- In cellForRowAtIndexPath muss dann das Team zum Index gefunden werden

```
    switch (self.view.tag) {
        case PG_GROUP_CONTROLLER:
            team = [[PGEntitySetup sharedInstance] fetchTeamWithIndexPath:indexPath]; break;
        case PG_TEAM_CONTROLLER:
            NSArray* teams = [[PGEntitySetup sharedInstance] fetchTeams];
            team = [teams objectAtIndex:indexPath.row];
            break;
    }
```

Die EM-App (Team-View)

- Für die Mannschaftsansicht wird ein neuer ViewController in das Storyboard eingebaut und direkt über den „Attribute Inspector“ benannt („*vc:details:team*“)
- Im Controller für die Gruppen-, bzw. Mannschaftenübersicht wird die Methode tableView:tableView didSelectRowAtIndexPath überschrieben, um bei einen Klick auf eine Tabellenzelle einen Übergang auf den Mannschaftsview einzuleiten

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    UIViewController* vcDetails = [self.storyboard instantiateViewControllerWithIdentifier:@"vc:details:team"];
    [self.navigationController pushViewController:vcDetails animated:YES];
}
```

- Für den neuen Controller wird nun noch eine neue Klasse (PGTeamViewController) angelegt und dieser mit einer Team-Property versehen, welche in der obigen Methode anhand des Indexpfades (siehe vorherige Folie) gesetzt wird
- Die Mannschaftsansicht soll über einen Header verfügen, in dem die Flagge, der Name und die Gruppenzugehörigkeit angezeigt wird, mittig soll eine Tabelle mit den Spielern und ganz untern Informationen über den Trainer angezeigt werden

Benutzerdefinierte Controller & Ansichten

- Im Gegensatz zum bisherigen Vorgehen wird für die Mannschaftsansicht kein bereits fertiger Controller genutzt, der nun erweitert wird, sondern es soll ein komplett eigener geschrieben werden
- Wer einen Blick auf die bisher eingesetzten Controller wirft, wird sehen, dass diese auch immer mit View-Pendant ausgestattet sind
- Also wird neben dem PGTeamViewController eine Klasse PGTeamView erstellt, die von UIView erbt und im Storyboard als View-Klasse angegeben wird
- Der neue View soll nach Voraussetzung über drei Teilbereiche (Header, Spielertabelle, Trainerinformationen) verfügen, die nun über das Storyboard designed werden können
- Der Einfachheit halber sollten diese drei Bereiche als IBOutlet im View verankert sein (wird dies nicht gemacht, muss man die drei Teilbereiche programmatisch hinzufügen, dazu später mehr)
- Für den Header- und Trainerbereich werden erneut UIView-Klassen erstellt und zugewiesen, während der Spielerbereich durch UITableView repräsentiert wird

Benutzerdefinierte Controller & Ansichten (II)

- Header und Spielerbereich können nun nach belieben über das Storyboard innerhalb des PGTeamViews designed werden
- Auch hier sollte man die einzelnen Bausteine über IBOutlets in den entsprechenden Klassen verankern, damit diese ansprechbar sind, sobald der View geladen wurde
- In der Methode können die beiden Bausteine dann anhand der Team-Instanz gefüllt werden, die zuvor durch das Einleiten des Übergangs vom vorherigen Controller gesetzt wurde:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    PGTeamView* view = (PGTeamView*) self.view;
    // header
    view.header.teamLabel.text = _team.name;
    view.header.flagView.image = [UIImage ... ]]];
    view.header.groupLabel.text = _team.group.name;
    // footer
    view.footer.coachLabel.text = [NSString stringWithFormat:@"%@ %@",   [_team.coach valueForKey: @"givenname"],
                [_team.coach valueForKey: @"name"]];
}
```

Benutzerdefinierte Controller & Ansichten (III)

- Jetzt fehlt noch den Spielerbaustein:
  - Da dieser eine Tabelle ist, sollte der zugehörige PGTeamViewController die nötigen Protokolle (UITableViewDelegate, UITableViewDataSource) erfüllen
  - Zudem sollen die Spieler aufgeteilt nach ihrer Position aufgelistet werden
  - Insgesamt existieren vier verschiedene Positionen: Torwart, Abwehr, Mittelfeld und Angriff

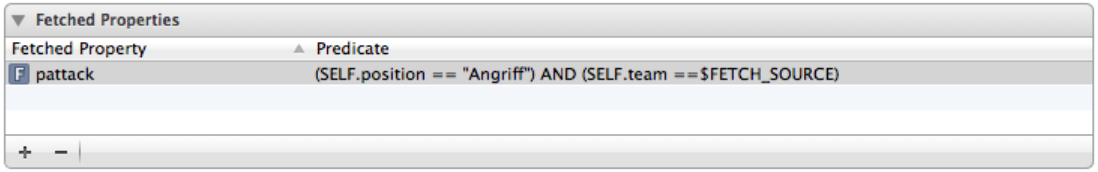
```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {return 4;}

-(NSString*) tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section {
    switch (section) {
        case 0: return @"Torwart";
        case 1: return @"Abwehr";
        case 2: return @"Mittelfeld";
        case 3: return @"Angriff";
    }

    return nil;
}
```

Fetched Properties

- In der Einleitung zu Core Data wurden „Fetched Properties“ bereits angesprochen:
  - „Fetched Properties“ sind Eigenschaften einer Entität die per Request an den ManagedObjectContext abgefragt werden
  - Zwischen einer solchen Eigenschaft und der Instanz herrscht also eine lose Kopplung, die vor allem nicht bidirektional sein kann
- Für die Spielertabelle werden solche „Fetches Properties“ nun genutzt, um für ein Team die Spieler auf einer bestimmten Position herauszufinden
- Die Eigenschaften werden also an die Entität „Team“ gekoppelt
- Zum Erstellen wird der Data-Model-Editor genutzt:



- Als „Destination“ muss die Klasse angegeben werden, von dessen Typ die Entitäten der Antwort sind (hier also „Player“)
- Als „Predicate“ muss in SQL der Request formuliert werden

Fetched Properties (II)

- Das Schüsselwort SELF bezeichnet hierbei die Entität vom Typ „Destination“, auf der das Predikät aktuell angewandt wird, während $FETCH_SOURCE die Entität bezeichnet, von der die Anfrage gestellt wurde
- Insgesamt wird in dem Beispiel also nach den Spielern in der Spielertabelle gesucht, deren Position „Angriff“ ist und in dem Team spielen, für das die Anfrage gestellt wurde
- Da die „Fetched Properties“ nachträglich in das Datenmodell eingefügt wurden, muss die Team-Klasse, um die Eigenschaften erweitert werden:

```
@property (nonatomic, retain) NSArray* pattack;
@property (nonatomic, retain) NSArray* pcenter;
@property (nonatomic, retain) NSArray* pdefense;
@property (nonatomic, retain) NSArray* pkeeper;
```
Team.h

```
@dynamic pattack;
@dynamic pcenter;
@dynamic pdefense;
@dynamic pkeeper;
```
Team.m

- Des weiteren kann es vorkommen, dass Änderungen am Datenmodell dazu führen, dass die Applikation nicht startet, weil das genutzte Modell nicht mit dem neuen übereinstimmt. In diesem Fall muss die App erst vom Gerät oder dem Simulator gelöscht werden

Fetched Properties (III)

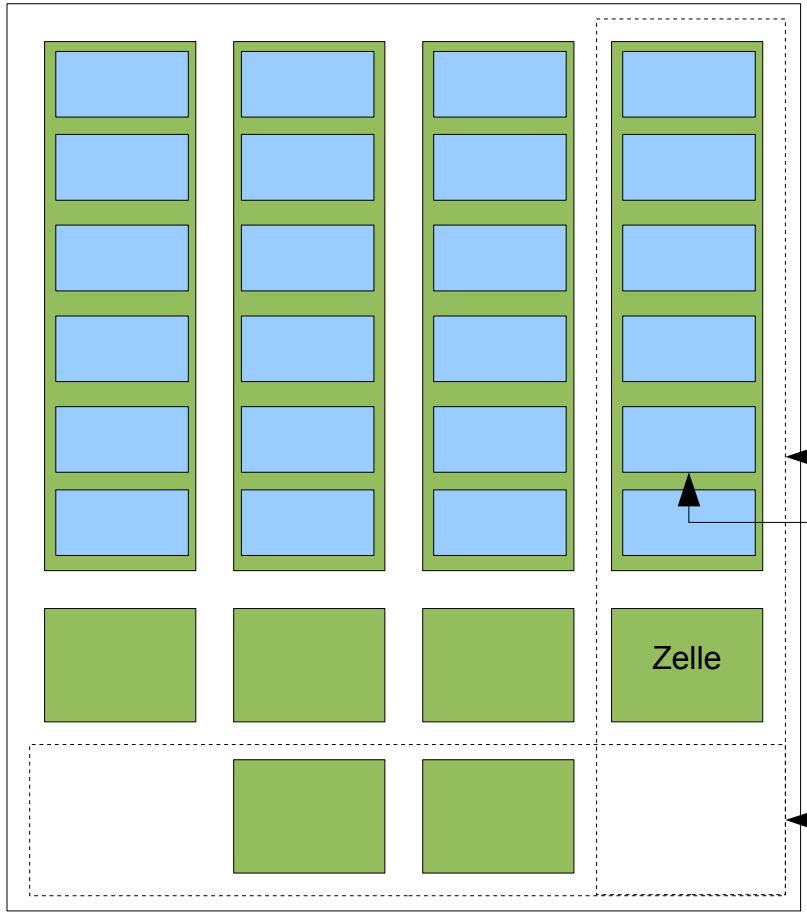
- Mit den „Fetched Properties“ können nun die noch fehlenden DataSource-Protokollmethoden implementiert werden:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    switch (section) {
        case 0: return [_team.pkeeper count]; case 1: return [_team.pdefense count];
        case 2: return [_team.pcenter count]; case 3: return [_team.pattack count];
    }
    return 0;
}
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    ...
    Player* player = nil;
    switch (indexPath.section) {
        case 0: player = [_team.pkeeper objectAtIndex:indexPath.row]; break;
        case 1: player = [_team.pdefense objectAtIndex:indexPath.row]; break;
        case 2: player = [_team.pcenter objectAtIndex:indexPath.row]; break;
        case 3: player = [_team.pattack objectAtIndex:indexPath.row]; break;
    }
    cell.textLabel.text = [NSString stringWithFormat:@"%@ %@",  player valueForKey: @"givenname"],  [player valueForKey: @"name"]];

    return cell;
}
```

Die Spielplanansicht

- Bei der Implementierung der Mannschaftsansicht wurde erstmals ein eigener View erstellt, der allerdings seinerseits auf bekannte Bausteine (TableView) zurückgriff
- Für die Spielplanübersicht soll nun eine komplett eigene Ansicht samt Protokoll definiert werden
- Die Ansicht ist in Spalten aufgeteilt, die wiederum zeilenweise in Zellen aufgeteilt sind. Zellen ihrerseits können „Datenzellen“ enthalten



Die Spielplanansicht (III)

- Zu erst wird eine UIView Klasse erzeugt: PGScheduleView

```
@interface PGSchedulerView : UIView

@end
```

- Wie gehabt erstellt man nun noch einen ViewController im Storyboard, eine neue UIViewController-Klasse, weist sowohl die Controller-Klasse als auch die PGScheduleView-Klasse im Storyboard entsprechend zu und verbindet den TabBarController mit dem neuen Controller
- Damit sind die Grundpfeiler gesetzt, allerdings wird in der Applikation auf der entsprechenden Seite bisher lediglich ein weißer Bildschirm angezeigt
- Die Spielplanansicht ist nach Definition aufgeteilt in Zellen und Datenzellen, deren Inhalte variieren, das prinzipielle Aussehen allerdings nicht
- Bisher würde man nun versuchen die Ansicht im Storyboard zu designend
- Der Ansatz schlägt hier allerdings fehl: Was benötigt wird ist ein Prototyp für die Zellen, der geladen und dann beliebig oft in den View integriert werden kann (vgl. UITableViewCell)

Bausteine über .xib-Vorlagen

- Storyboards sind neu im Cocoa-Framework
- Zuvor wurden Oberflächen über den Interface-Builder erstellt und in sogenannten „nib“-Dateien gespeichert
- xib sind die XML-basierte Variante der .nib-Dateien
- Obwohl Storyboard den etwas intuitiveren Ansatz für das Designen eins Frontends bieten, kann auch weiterhin der Interface-Builder genutzt werden
- Für die Spielplanansicht wird er nun dazu benötigt werden, Prototypen für die Zellen zu definieren, die dann vom PGScheduleView erzeugt werden, um die Ansicht zu generieren
- Eine neue .xib-Datei wird über „File“ → „User Interface“ → „Empty“ angelegt
- Per Klick öffnet sich der Interface-Builder, dessen Oberfläche dem Storyboard ähnelt
- Für die Hauptzellen zieht man einen UIView in den „Sketch“-Bereich und ändern über den „Attribute Inspector“ die Hintergrundfarbe

Bausteine über .xib-Vorlagen (II)

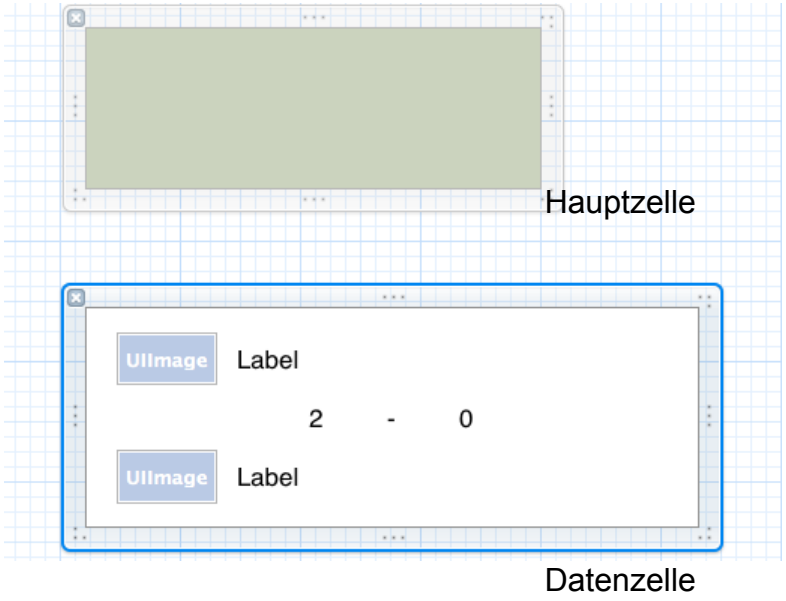
- Die Vorlage lässt sich nun ganz einfach einlesen:

```
NSArray* array = [[NSBundle mainBundle] loadNibNamed:@"XIB-NAME" owner:self options:nil];
```

- Der Name des Nib muss ohne die Dateiendung angegeben werden
- Der „Owner“ bezeichnet das Objekt, das verantwortlich für die in der Vorlage definierten Bausteine sein wird (später mehr)
- Wie man sieht, liefert der Aufruf einen Array; d.h. zum einen können in der Vorlage mehrere Bausteine definiert werden, zum anderen aber, dass es eine Möglichkeit geben muss, die Bausteine auseinander zu halten
- Die einfachste Möglichkeit ist es, die Bausteine über den „Attribute Inspector“ mit einem Tag zu versehen
- Wie schon im Beispiel zum „geteilten ViewController“ (Folie 7ff.) sollte man der besseren Lesbarkeit halber ein Enum definieren
- Mit Hilfe der Tags kann nun durch die Array iteriert und nach einem bestimmten Baustein gesucht werden
- Da dieser Vorgang immer gleich ist, sollte man ihn in eine eigene Klasse auslagern

Bausteine über .xib-Vorlagen (III)

- Die Vorlage für die Hauptzellen war relativ trivial
- Nun kann die Vorlage für die Datenzellen definiert werden, die im Spielplan ein bestimmtes Spiel repräsentieren
- Natürlich darf nicht vergessen werden, die neue Vorlage entsprechend zu „taggen“
- Da der Baustein Daten darstellen soll, empfiehlt es sich die Teilbausteine, die variierende Daten enthalten in eine eigene UIView Klasse zu kapseln und per IBOutlet zu verbinden
- Dies geschieht im Interface-Builder analog zum Storyboard: Zu erst den Klasse im „Identity Inspector“ zu weisen und anschließend die Outlets mit den Teilbausteinen verbinden



Das PGScheduleDelegate-Protokoll

- Mit den beiden Vorlagen für die Zellen könnte der PGScheduleView nun sobald er geladen wurde die Ansicht aufbauen
- Allerdings benötigt er dazu noch die Daten, die er über das Delegate erlangen soll
- Ganz nach dem MVC-Pattern sollte man natürlich eine klare Trennung zwischen Ansicht und Controller anstreben
- In den bisher behandelten View und Controller-Beispielen wurde diese Trennung über Delegates erreicht, über die eine UIView-Instanz, die zum Aufbau der Ansicht nötigen Daten sammeln konnte (siehe z.B. UITableViewDataSource)
- Ein Delegate sollte immer einem bestimmten Protokoll genügen, weshalb man zu erst solches anlegt und die PGScheduleView-Klasse um ein Outlet erweitert:

```
@interface PGScheduleView : UIView
@property(nonatomic, assign) IBOutlet id<PGScheduleDelegate> delegate;
@end
```

- Das neue Outlet kann dann über das Storyboard mit dem PGScheduleController verbunden werden, der zudem um das neue Protokoll erweitert wird

Das PGScheduleDelegate-Protokoll (II)

- Folgende Daten werden zum Aufbau der Ansicht benötigt
  - Maximale Anzahl an Spalten
  - Maximale Anzahl an Zeilen
  - Anzahl der Spalten in einer bestimmten Zeile
  - Anzahl der Zellen innerhalb einer Spalte
  - Die Daten des Spiels zu einer bestimmten Datenzellen
- Daraus ergibt sich folgender erster Ansatz für ein Protokoll:

```
-(NSUInteger) numberOfColumnsInSchedulerView: (PGScheduleView*) view;                                   (A)
-(NSUInteger) numberOfRowsInScheduleView: (PGScheduleView*) view;                                       (B)
-(NSUInteger) scheduleView: (PGScheduleView*) view numberOfColumnsInRow: (NSUInteger) column;           (C)
-(NSUInteger) scheduleView:(PGScheduleView*) view numberOfCellsInColumn: (NSUInteger) column;           (D)
-(Match*) scheduleView: (PGScheduleView*) view matchForIndex: (PGCellIndex*) cellIndex;                 (E)
```

- Desweiteren werden Zusatzinformationen für bestimmte Spiele benötigt, bspw. was in den Playoffs angezeigt werden soll, wenn die Gegner noch nicht feststehen

```
-(NSDictionary*) scheduleView: (PGScheduleView*) view detailsForIndex: (PGCellIndex*) indexPath;       (F)
```

- Zudem wird auch das Füllen einer Datenzelle an das Delegate gereicht

```
-(void) setup: (PGScheduleSubcell*) cell withMatch: (Match*) match andDetails: (NSDictionary*) details   (G)
```

Protokoll-Implementierung

- Der Ansatz der Spielplanimplementierung ist möglichst allgemein gehalten
- Für den EM-Plan ergibt sich:
  - Es gibt insgesamt 33 Spiele aufgeteilt, wobei die Gruppenspiele und die Playoffs jeweils als Block dargestellt werden sollen
  - Es sind also 4 (A) Spalten
  - 4 Zeilen (B)
  - In den ersten beiden Zeilen werden jeweils vier Spalteneinträge, im Halbfinale zwei und im Final einer erwartet (C)
  - In der erste Zeile werden zudem sechs Datenzellen ansonsten eine erwartet (D) (hier wird davon ausgegangen, dass die Anzahl pro Spalte nicht variiert)
  - Die Match-Instanz zu einer bestimmten Zelle erhält man per Mapping des Zellenindices auf das „Group“-Attribute in jedem Spiel (E)
  - Als Details wird das NSDictionary aus der setup.plist zu einem bestimmten Spiel herausgesucht (F)
  - Die setup-Methode (G) füllt letztendlich die Datenzellen