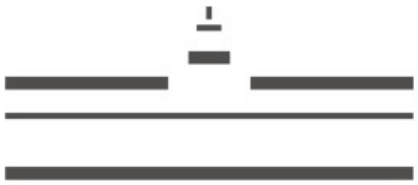


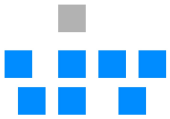
Westfälische
Wilhelms-Universität
Münster

Programmierung für mobile Endgeräte

Core Data

wissen.leben
WWU Münster

Name: Patrick Förster



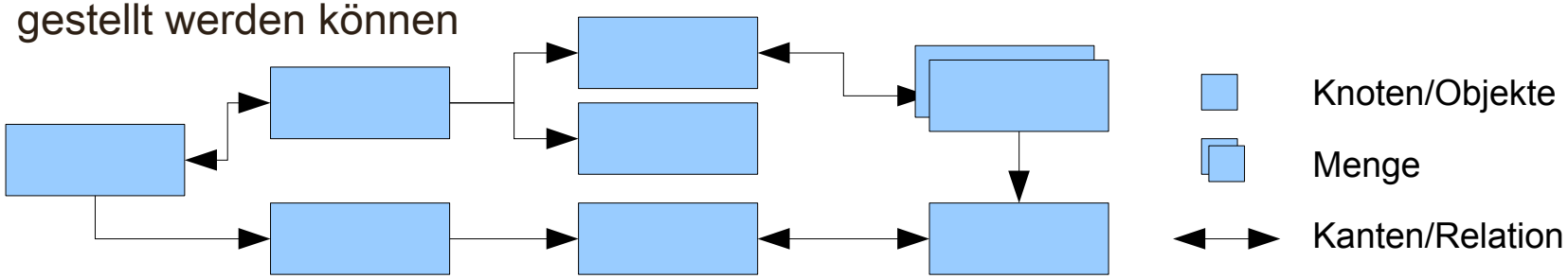
ZENTRUM FÜR
INFORMATIONS
VERARBEITUNG

Motivation

- Bisher wurden pList-Dateien als Quellen genutzt, um Daten an eine Applikation zu binden
- Für einfache Datensätze, wie die Kennzeichen → Stadt/Landkreis Zuordnung ist dieser Ansatz auch durchaus vertretbar
- Sobald die Daten komplexer werden und insbesondere Relationen zwischen den Daten modelliert werden sollen, würde der bisherige Ansatz aufwendig und fehleranfällig werden
- Hinzukommt, dass bisher lediglich ein lesender Zugriff auf Daten nötig war
- Eine Interaktion mit den Daten, in etwa im Sinne von Änderungen war bislang ebenso wenig Thema, wie das Persistieren der Daten, so dass diese das Beenden einer Applikation überleben
- Ideal wäre also ein Framework, dass die Relation von Objekten abbildet, die Objekte verwaltet und ggf. auch dafür Sorge tragen kann, Objektdaten zu speichern

Core Data

- Das „Core Data“ Framework übernimmt genau diese Aufgaben
- Wichtig hierbei ist: Core Data ist kein Datenbanksystem!
- Core Data stellt anhand der Beschreibung bestimmter Daten einen Objekt-Graphen zusammen, dem ähnlich wie in einem Datenbanksystem Anfrage gestellt werden können



- Es ist aber keinesfalls vorgeschrieben, dass der Graph persistent sein muss:

> Core Data is not a relational database or a relational database management system (RDBMS).
>
> Core Data provides an infrastructure for change management and for saving objects to and retrieving them from storage. It can use SQLite as one of its persistent store types. It is not, though, in and of itself a database. (To emphasize this point: you could for example use just an in-memory store in your application. You could use Core Data for change tracking and management, but never actually save any data in a file.)
>
> Apple Dokumentation: Core Data

Objektgraph ↔ Datenbank

- Datenbanken sind auf den Zugriff auf bestimmte Tabelleneinträge optimiert
- Datensätze müssen eine feste Größe haben
- Im Gegensatz zu einem Objektgraphen verweist daher nicht das Vaterobjekt auf seine Kinder, sondern die Kinder auf die Väter:

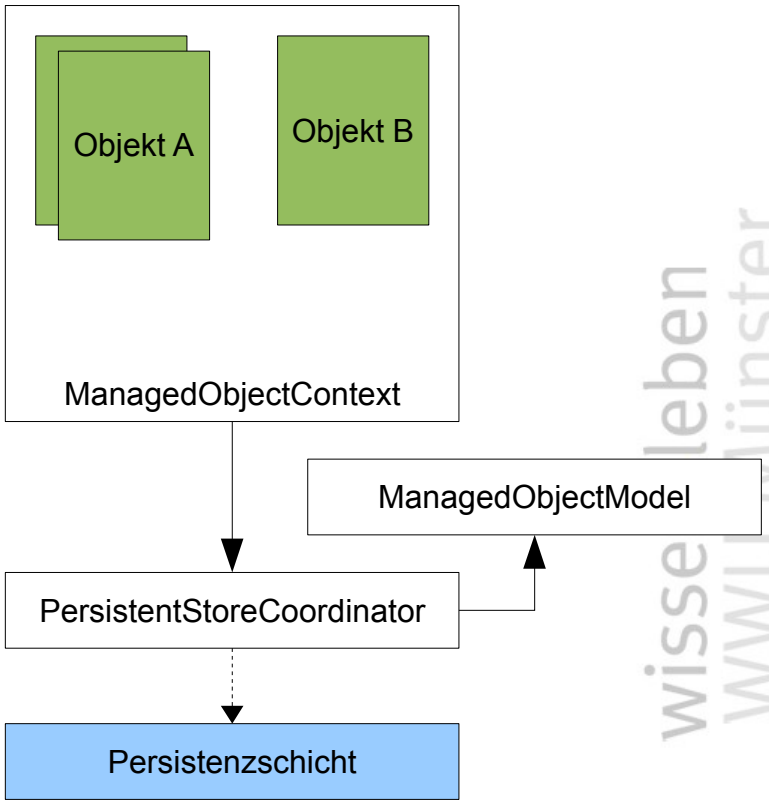
ID	Name	...
505	Jakob	...

ID	VID	Name	...
0	505	Ruben	...
1	505	Simeon	...
...	...	...	...

- Wenn die Tabelle n Väter und m Kinder hat und man möchte auch nur zu einem Vater alle Kinder finden, muss immer die gesamte Tabelle, also m Datensätze durchsucht werden
- In einem Graphen ist ohne Suche zu jedem Vater direkt bekannt, wie viele Kinder er hat!

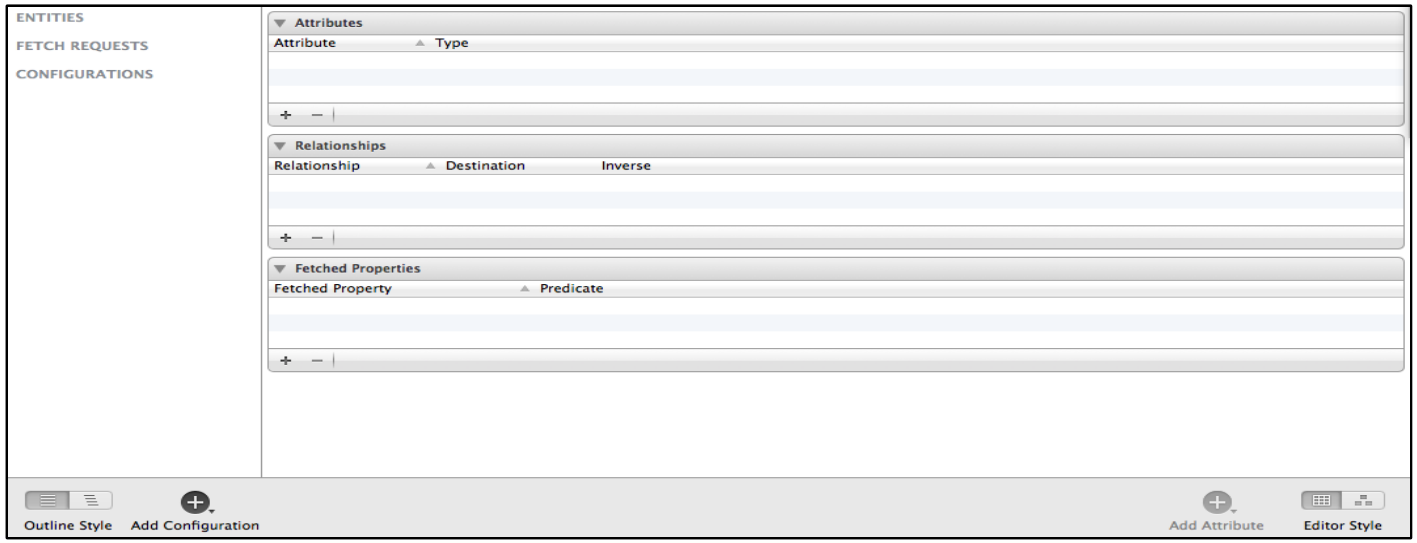
Core Data: Aufbau

- Das Grundgerüst von Core Data basiert auf den folgenden drei Bausteinen:
- NSManagedObjectContext: Der Kontext beinhaltet und verwaltet alle aktuell benötigten Core-Data-Objekte („Managed Objects“)
- NSPersistentStoreCoordinator: Dieser Baustein ist die Schnittstelle zwischen dem Objektkontext und einer evtl. genutzten Persistenzschicht und sorgt dafür, dass aus persistierten Daten „Managed Objects“ werden und umgekehrt
- NSManagedObjectModel: Durch das Objektmodell werden die Objekte beschrieben, die durch den Koordinator gelesen bzw. gespeichert werden können



Core Data: Objekt-Modell

- Bei einer Anwendung mit Persistenzbedarf sollte im ersten Schritt immer das Objekt-Model definiert werden
- Xcode bietet hierfür eine komfortable Oberfläche
- Per „New File“ → „Core Data“ (links) → „Data Model“ (Template) wird ein leeres Datenmodell angelegt
- Per Klick auf die angelegte Datei öffnet sich der Datenmodell-Editor:



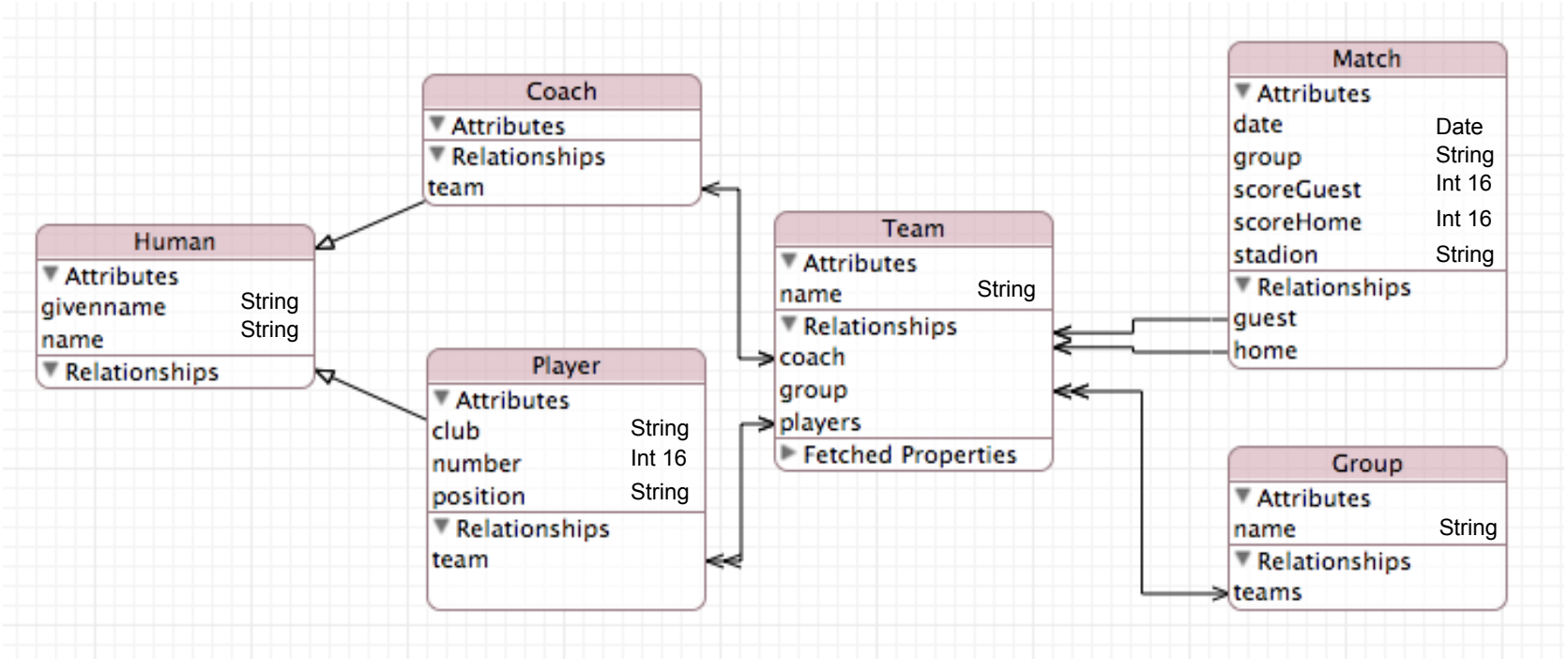
Core Data: Objekt-Modell

- *Entites*: Eine Entität ist ein Objekt, das aus der Persistenzschicht in den Kontext übertragen werden kann:
  - *Attributes*: Attribute entsprechen primitiven (Number, String, Boolean, ect.) Objekt-Eigenschaften („*Properties*“)
  - *Relationships*: Beziehungen zwischen Objekten werden durch Referenzen auf die entsprechenden Objekte abgebildet.
    - Beziehungen können mehrdimensional sein, d.h. ein Objekt kann durchaus mehrere Objekte gleichen Typs referenzieren; aus der einfachen Referenz wird dann ein NSSet mit Referenzen
    - Referenzen können bidirektional sein; dafür muss in der referenzierten Klasse eine Möglichkeit der inversen Referenzierung gegeben sein
  - *Fetched Properties*: Im Gegensatz zu „Relationships“ sind „Fetched Properties“ Referenzen auf Daten, die erst durch eine Anfrage (siehe „*Fetch Request*“) an den Objekt-Kontext gesetzt werden

Core Data: Objekt-Modell (II)

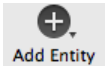
- *Fetch Request*: Ein Request ist eine in SQL-Syntax gestellte Anfrage an den Objekt-Kontext Daten mit bestimmten Eigenschaften zu liefern. Unter „Fetch Request“ können solche Anfrage über die Oberfläche per Klick zusammengestellt werden; in einfachen Fällen ganz ohne explizites SQL
- *Configurations*:
  - Über die Entitäten werden die Daten beschrieben, die sich in der Persistenzschicht befinden (In einem Datenbanksystem würde man hier von Tabellen reden)
  - Der PersistentStoreCoordinator wandelt diese Daten zur Laufzeit in „Managed Objects“ um. Objekte werden allerdings nicht durch das Modell sondern durch eine Klasse definiert, d.h. es muss noch eine Verbindung zwischen Entität und Klasse festgelegt werden.
  - In der Konfiguration wird festgelegt welchen Typ eine Entität als ManagedObjectContext haben wird

Core Data: EM-Planer Modell



- Xcode erzeugt automatisch basierend auf dem Datenmodell einen Graphen, der alternative zur tabellarischen Ansicht genutzt werden kann, um das Modell zu bearbeiten
- Zum Ändern der Ansicht im Modell-Editor unten rechts auf klicken

Core Data: EM-Planer Modell (Attribute)

- Core-Data beherrscht Vererbung im Entitätenmodell
- Zur Demonstration dient die „Human“ Entität mit ihren beiden Nachkömmlingen „Coach“ und „Player
- Per 

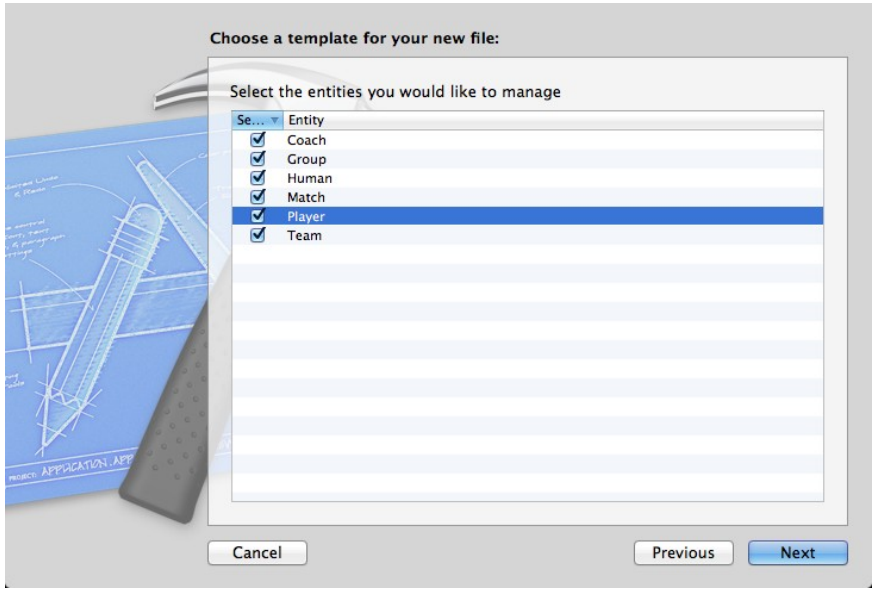
wird eine neue Entität erzeugt und umgehend benannt
- Über das + in der Attribut-Tabelle werden nun die zwei Attribute „givenname“ und „name“ hinzugefügt und der Typ auf „String“ eingestellt
- Im „Attribute Inspector“ (rechts) wird nun noch ein Häkchen bei „Abstract Entity“ gemacht, da im Programm später nur Entitäten vom Typ „Coach“ oder „Player“ instanziiert werden sollen
- Analog werden anhand des Modells die Entitäten „Coach“ und „Player“ erzeugt
- Über den „Attribute Inspector“ setzt man die „Parent Entity“ auf „Human“
- Nun können die restlichen Entitäten erzeugt werden
- Jetzt fehlen noch die Beziehung der Entitäten untereinander:
  - Diese sind im Modell als Verbindungen dargestellt, wobei eine einfache Spitze eine einfache und eine doppelte eine mehrfache Beziehung darstellt

Core Data: EM-Planer Modell (Beziehungen)

- Angefangen bei der Entität „Team“ fügt man über das + in der „Relationships“-Tabelle eine neue Beziehung hinzu, benennt diese wie im Modell angegeben und setzt den „Destination“ Eintrag auf die entsprechende Klasse
- Solange in der „Destination“ Entität noch keine inverse Beziehung definiert worden ist, lässt sich diese in der Tabelle auch noch nicht setzen
- Also fügt man der „Coach“ Entität eine Beziehung zur „Team“ Entität hinzu, konfiguriert (sieheModell) und setzt dann den Eintrag zu „Inverse“
- Der Editor erkennt die inverse Beziehung und setzt das Gegenstück in der „Team“ Entität automatisch
- Das Datenmodell besagt, dass zwischen der „Team“ und der „Player“ Entität eine 1:n Beziehung besteht, d.h. jedes hat mehrere Spieler, jeder Spieler gehört aber nur zu einem Team
- Um dies Abzubilden muss im „Attribute Inspector“ für die Spielerbeziehung in der „Team“ Entität ein Häkchen bei „To-Many Relationship“ gemacht werden
- Alle anderen Beziehungen können nun analog erstellt werde

Core Data: EM-Planer Modell (Klassen)

- Das Datenmodell der Persistenzschicht ist damit fürs Erste definiert
- Für den Kontext fehlen aber noch die zu den Entitäten gehörenden Klasse
- Diese können sich einfach durch Xcode erzeugen lassen:
  - Im Projekt-Navigator andere als die Datenmodell-Datei auswählen (ansonsten wird nur die Klasse für die aktuell markierte Entität generiert)
  - „New File“ → „Core Data“ → „NSManagedObject subclass“
  - Im nächsten Dialog das gewünschte Datenmodell auswählen



- Abschließend den Zielordner auswählen und Xcode den Quellcode generieren lassen

Core Data: EM-Planer Modell (Klassen II)

- Ein Blick in die generierten Klassen lohnt sich:
  - Alle Klassen erben von NSManagedObject, also insb. auch „Player“ und „Coach“, die angegebene Vererbung von „Human“ ist also auf den ersten Blick verschwunden (dazu später mehr)
  - Alle erzeugten Properties werden in der Implementierung nicht mit @synthesize sondern mit @dynamic angegeben. Dies veranlasst zum einen dass die Getter/Setter-Methoden nicht generiert werden, zum anderen wird dem Compiler aber mitgeteilt, dass die Methden durchaus existieren (hier in der Oberklasse: NSManagedObject)
  - In den Klassen mit „to Many“ Beziehung werden sogenannte „Kategorien“ erstellt: Durch Kategorien können bestehende Klassen um Methoden erweitert werden ohne eine Unterklasse zu erzeugen

```
@interface Team (CoreDataGeneratedAccessors)

- (void)addPlayersObject:(NSManagedObject *)value;
- (void)removePlayersObject:(NSManagedObject *)value;
- (void)addPlayers:(NSSet *)values;
- (void)removePlayers:(NSSet *)values;

@end
```

Core Data: Erzeugung der drei Bausteine

- Das Objektmodell wird im Programm durch NSManagedObjectModel repräsentiert
- Die Klassenmethode mergedModelFromBundle veranlasst, dass alle Datenmodellbeschreibungen in ein Objektmodell geladen werden

```
_managedObjectModel = [NSManagedObjectModel mergedModelFromBundles:nil];
```

- Das Datenmodell wird von einem NSPersistentStoreCoordinator benötigt:

```
_persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
                               initWithManagedObjectModel:[ managedObjectModel]];

NSURL* storeURL = [[self applicationDocumentsDirectory] URLByAppendingPathComponent:@"em.sqlite"];
NSError* error;
if (![_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType configuration:nil
                                                       URL:storeURL options:nil error:&error]) {...}
```

- Hier wird ein zuerst ein Koordinator auf dem zuvor erzeugten Datenmodell erstellt
- Nachfolgend wird eine Store basierend auf einer SQLite-Datenbank hinzugefügt
- Falls kein Store mit dem Namen existiert, wird ein neuer angelegt. Die Benennung der Datenbank ist im Prinzip beliebig, da nie über außer bei der Initialisierung über den Namen auf die Dankbank zugegriffen wird

Core Data: Erzeugung der drei Bausteine (II)

- Abschließend fehlt nur noch der NSManagedObjectContext

```
_managedObjectContext = [[NSManagedObjectContext alloc] init];
[_managedObjectContext setPersistentStoreCoordinator:persistentStoreCoordinator];
```

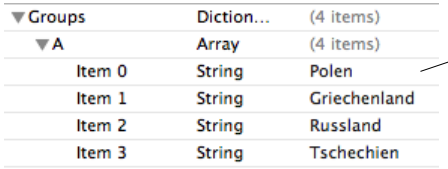
- Da die Bausteine nur einmalig erzeugt werden müssen und um einen programmweiten Zugriff zu haben, werden sie ins AppDelegate eingebaut:

```
@property (strong, nonatomic) NSManagedObjectModel* managedObjectModel;
@property (strong, nonatomic) NSPersistentStoreCoordinator* persistentStoreCoordinator;
@property (strong, nonatomic) NSManagedObjectContext* managedObjectContext;
```

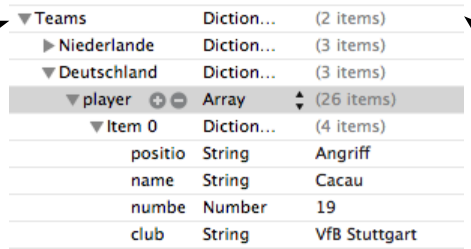
- Die Getter-Methoden werden entsprechend der vorherigen Folien implementiert
- Damit diese allerdings compilieren, muss Core Data dem Projekt als Framework hinzugefügt werden („Project Navigator“ → Projekt → „Summary“ → „Frameworks“)
- Als Test kann man in der application:didFinishLaunchingWithOptions Methode den Objektkontext aufrufen und den Simulator starten
- Mit einem entsprechenden SQLite-Programm lässt sich die Datenbank betrachten (zu finden unter: */Users/USER/Library/Application Support/iPhone Simulator/5.1/Applications/APPLICATION-ID/Documents*)

Core Data: Initialisierung der Datenbank

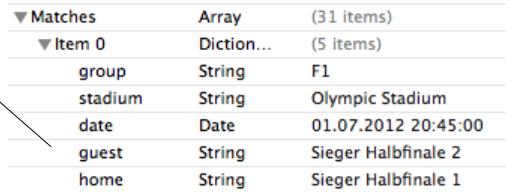
- Leider gibt es in Xcode keine Möglichkeit die Datenbank initial zu füllen
- Man könnte probieren, mit einem Datenbank-Browser die erzeugte Datenbank zu füllen, allerdings ist bei erster Betrachtung a priori nicht klar, welche Werte unter „Z_ENT“ oder „Z_OPT“ erwartet werden
- Wenn es im Programm eh vorgesehen ist, Daten ändern zu können, könnte man das eigene Programm nutzen um eine Initialdatenbank zu erstellen, die später im Produkt mitgeliefert wird
- Für die EM-Plan Applikation wird der bereits beschrittene .pList-Pfad genommen
- Die „setup.pList“ Datei einhält einen kleinen Teildatensatz, so dass Oberfläche der Applikation designed und getestet werden kann



Gruppen + Mannschaften



Mannschaften + Spieler + Trainer



Spiele + Mannschaften

Core Data: Initialisierung der Datenbank (II)

- Der Initialisierungsprozess verläuft folgendermaßen
  - Zu erst werden per Iteration durch den Gruppeneintrag alle Gruppen erzeugt
  - Im Zuge der Gruppenerzeugung werden alle Mannschaft erstellt, allerdings werden die Objekte vorerst nur den Namen der Mannschaft kennen
  - Per Iteration durch den Mannschaftseintrag werden die bereits erstellten Mannschaften dann mit den Spielern und dem Trainer gefüllt, in dem diese durch die Angaben in den Unterbäumen erzeugt werden werden
  - Abschließend wird dann per Iteration der Spielplan erzeugt
- Das Erzeugen einer Entität verläuft immer auf die gleiche Art und Weise:

```
NSManagedObjectContext* ctx = [self managedObjectContext];
ENTITY* entity = [NSEntityDescription insertNewObjectForEntityForName:@"ENTITY" inManagedObjectContext:ctx];
entity.number = ...
```

- Mit Hilfe der Klassen-Methode insertNewObjectForEntityForName wird anhand des Klassennamens im angegebenen Kontext ein neues „Managed Object“ erzeugt, dessen Attribute dann auf herkömmliche Art und Weise gesetzt werden können

Core Data: Initialisierung der Datenbank (III)

- Spezielle Aufmerksamkeit ist dabei den Entitäten „Coach“ und „Player“ zu widmen:
  - Nach Definition erben diese von „Human“, allerdings ist dies nicht in der Klassenhierarchie wiederzufinden
  - Die geerbten Attribute „name“ und „givenname“ können also nicht direkt gesetzt werden
- Hier hilft die Methode setPrimitiveValue der Basisklasse NSManagedObject:

```
NSManagedObject* player = [NSEntityDescription insertNewObjectForEntityForName:@"Player" inManagedObjectContext:ctx];
[player setPrimitiveValue:@"VORNAME" forKey:@"givenname"];
[player setPrimitiveValue:@"NACHNAME" forKey:@"name"];
```

- Wenn ein Objekt in eine „to Many“ Relation eingefügt werden soll, muss dies über die erzeugten Kategorie-Methoden geschehen:

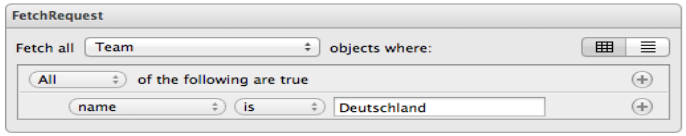
```
Player* player = ...
Team* team = ...

[team addPlayersObject:player];
```

Core Data: Initialisierung der Datenbank (IV)

- Abschließend sollte die Initialisierung das Speichern der Daten veranlassen

```
NSError *error;
if (![[self managedObjectContext] save:&error]) {...}
```

- Die .pList-Date soll nur zur Initialisierung der Datenbank genutzt werden, falls noch keine existiert
- Würde die Routine bei jedem Start durchgeführt werden, würden Änderungen die über das Programm vorgenommen wurden evtl. überschrieben werden
- Um zu Überprüfen ob die Datenbank bereits gefüllt wurde, wird ein „Fetch Request“ angelegt



- Der Request bietet darum, alle Mannschaften mit Namen „Deutschland“ zu liefern
- Durch Ausführen des Request kann also überprüft werden, ob die Datenbank bereits initialisiert wurde:

```
NSFetchRequest* request = [[self managedObjectModel] fetchRequestTemplateForName:@"initialisationCheck"];
NSError *error = nil;
NSArray *array = [self managedObjectContext executeFetchRequest:request error:&error];
if (array == nil) {…}
return [array count] != 0;
```