

CoreData

Programování zařízení Apple (IZA)

Martin Hrubý, FIT VUT, 2022/23

Motivační úvod

- Data a jejich správa je typicky cílem/smyslem aplikací.
- Nad daty konstruujeme algoritmy, tj funkcionalitu.
- Data uživatele musí být "v suchu a bezpečí" :)

V okamžiku formulování datového schématu (model) aplikace typicky rozhodujeme o jejím osudu :)

Data uživatele v ekosystému zařízení (Apple, mimo něj).

Ukládání dat v aplikacích

- Soubory s vlastními formáty. (dokumenty)
- UserDefaults.
- Dokumenty spravované iOS.
- Databáze.

Objektový kontejner - objektová paměť (OO databáze, Gemstone).

- Třída/Entita. Reference mezi objekty.
- Technická implementace (Sqlite3, XML/JSON, ...).

Alternativy

- Realm.
- Dropbox DB-API. Google Cloud (Firebase).
- DynamoDB (Amazon AWS).
- Cloudkit.
- CoreData+Cloudkit.

Lokální DB. Synchronizační prostředek.

Ekosystém aplikací: synchronizace uživatelských dat.

Co je CoreData (CD)

- Objektový kontejner.
- Rozhraní k uložišti persistentních objektů:
 - Vkládání a rušení objektů.
 - Správa vazeb mezi objekty.
 - Dotazy (fetch).

CD je z Foundation, tj. přenositelnost kódu na macOS (až na FRC). SwiftUI.

CD není relační DB

Připomenutí relačního kalkulu:

- projekce - `select ... from XY`, nelze.
- selekce - `select * from XY where ...`, LZE.
- join - `select ... from XY, YZ where ...`, nelze.

CD je pouze kolekce objektů (různých Entit/Tříd).

- výsledkem DB operace není relace, ale pole objektů.

Atributy Entity

Entity-Relation.

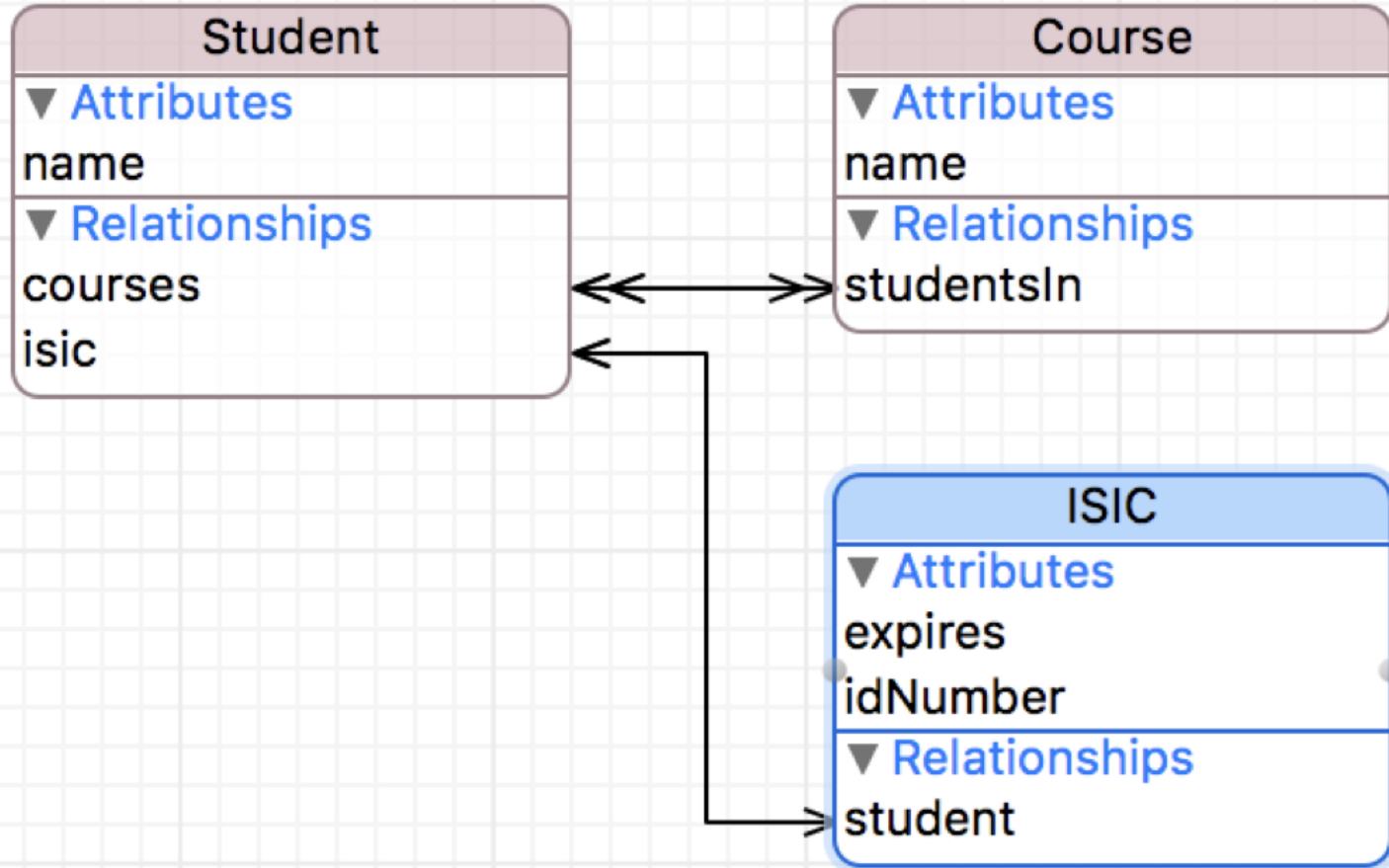
- Uložené atributy — název, datový typ.
- Relationships — název, typ (1:1, 1:N, M:N), cíl, inverzní vztah.
- Fetched Properties — ????.

Dědičnost ve schématu DB:

- Abstraktní entita — nemá uloženou podobu (tabulku), bude sloužit jako "parent".
- Parent entity (dědičnost entit).

RelationShips tam sice jsou, ale spíše obezřetně.

XCode DB Model Editor



XCode DB Model Editor

Pojem Entita se v kódu aplikace promítne do třídy (class).

The screenshot displays the XCode DB Model Editor interface. On the left, a sidebar lists entities (Course, ISIC, Student), fetch requests (allStudents), and configurations (Default). The main area is divided into three sections: Attributes, Relationships, and Fetched Properties. The 'Relationship' section on the right provides detailed configuration for the 'courses' relationship, including name, properties, destination, inverse, delete rule, type, arrangement, count, and advanced options like 'Index in Spotlight' and 'Store in External Record File'.

ENTITIES

- Course
- ISIC
- Student

FETCH REQUESTS

- allStudents

CONFIGURATIONS

- Default

Attributes

Attribute	Type
S name	String

Relationships

Relationship	Destination	Inverse
M courses	Course	studentsIn
ISIC	ISIC	student

Relationship Configuration (courses)

Name: courses

Properties: Transient Optional

Destination: Course

Inverse: studentsIn

Delete Rule: Nullify

Type: To Many

Arrangement: Ordered

Count: Unbounded (Minimum/Maximum)

Advanced: Index in Spotlight

Deprecated: Store in External Record File

User Info

Key	Value
-----	-------

Versioning

Hash Modifier: Version Hash Modifier

Renaming ID: Renaming Identifier

Objektová DB

Význam referencí mezi objekty:

- Nejsou režijní atributy relačního modelu jako klíče pro spojování záznamů (objektově orientované DB — GemStone).
- 1:1 — Student "má" ISIC,
- 1:N — Student "má zapsáno" [Course].
- 1:N — Course "má" [Student]. V důsledkům M:N.
- Ukládání referencí je ve správě/režii CD.
 - Obecně transformace DB-Modelu na DB-schema.

Synchronizace. Integrita DB.

Co nás bude zajímat

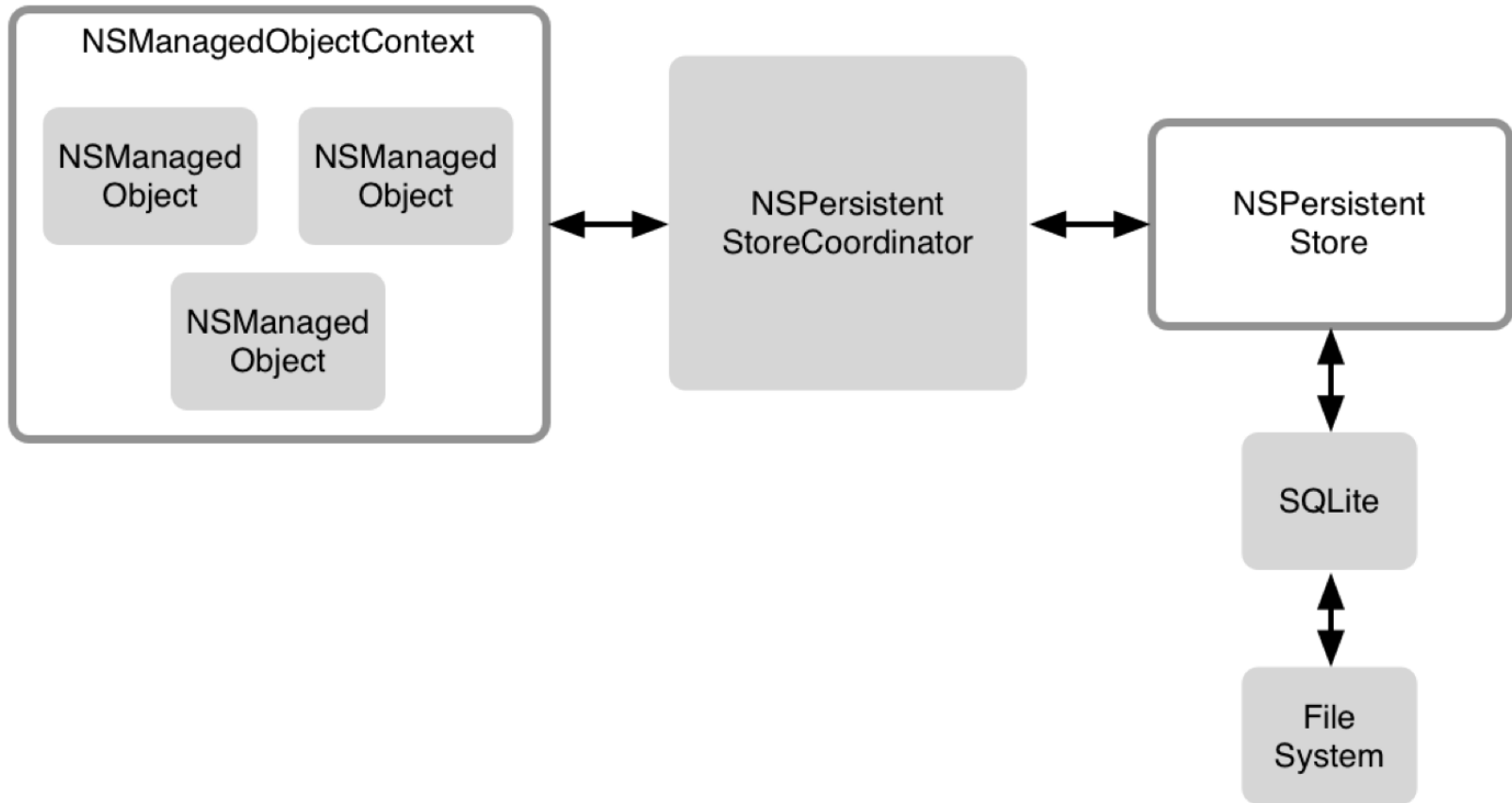
- Architektura CoreData knihovny.
- Formulace datového modelu aplikace.
- Způsob integrace CD do aplikace.
- Způsob fungování CD.
- Synchronizovaná CD (Cloud) — ve spojení s CloudKit.
- Návaznost na: UIDocument (kódování dat), CloudKit.

Architektura CD

CD je knihovna z Foundations, implementovaná v Objective-C.

- CD staví na principu *Key-Value Coding*, tj. pracujeme s NSObject (Foundation).
- NSManagedObject — nadtřída pro persistentní objekt.
- NSManagedObjectModel — datový popis DB Schematu.
- NSManagedObjectContext — OO paměť.
- NSPersistentStoreCoordinator — řadič na fyzická uložení.
- NSPersistentContainer — svazek významných objektů (model, MOC, StoreCoordinator).

Typická infrastruktura CoreData



NSManagedObjectModel

- NSEntityDescription, NSPropertyDescription.
- Inicializuje se z URL v App-Bundle (spec formát).
- Verzování:
 - Automatické — "Add model version..." (XCode)
 - Manuální — stará/nová DB, transformace objektů.

V XCode editujeme specializovaný dokument typu CoreData:Data model. Ten se stává součástí app bundle.

Verzování DB modelu je samostatná kapitola.

NSPersistentStoreCoordinator

- Konstrukce: objekt s DB Modelem. Drží model.
 - Umožňuje transformace DB (verzování).
 - V podstatě řízená kopie dat z jedné DB/Modelu do druhé DB/ Modelu.
- Vykonává (serializuje) DB operace.
- NSPersistentStore — ovladač nad konkrétním uložištěm (SQLite3, XML, další).
- Vlastní uživatelské ovladače na další typy uložišť.

Sqlite3 = základní implementace .DB pro všechny potřeby SW Apple.

Architektura celkově

- Instancuje se CD-Model.
- Nad CD-Modelem se instancuje StoreCoordinator.
 - Přidají se PersistentStores.
- Nad StoreCoordinatorem se konstruuje ManagedObjectContext (MOC).
- MOC je zapouzdřen v *PersistentContainers* - **CoreData Stack**.

ManagedObjectContext (MOC):

- Obvykle singleton v aplikaci (main-thread MOC).
- (MOCů lze mít v aplikaci více...). Background tasks.

Uložené datové typy

- String
- Různé typy čísel (Int, Double). Bool.
- Date.
- Binary Data (BLOB). Příloha. Zakódovaný dokument.

CoreData v XCode

XCode nám do projektu vygeneruje kód pro zapojení CD.

```
import CoreData
///
struct PersistenceController {
    //
    static let shared = PersistenceController()
    //
    let container: NSPersistentContainer
    //
    init(inMemory: Bool = false) {
        container = NSPersistentContainer(name: "cd1")

        //
        if inMemory {
            container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }

        //
        container.viewContext.automaticallyMergesChangesFromParent = true
        container.loadPersistentStores(completionHandler: { (storeDescription, error) in
            if let error = error as NSError? {
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
    }
}
```

... v XCode

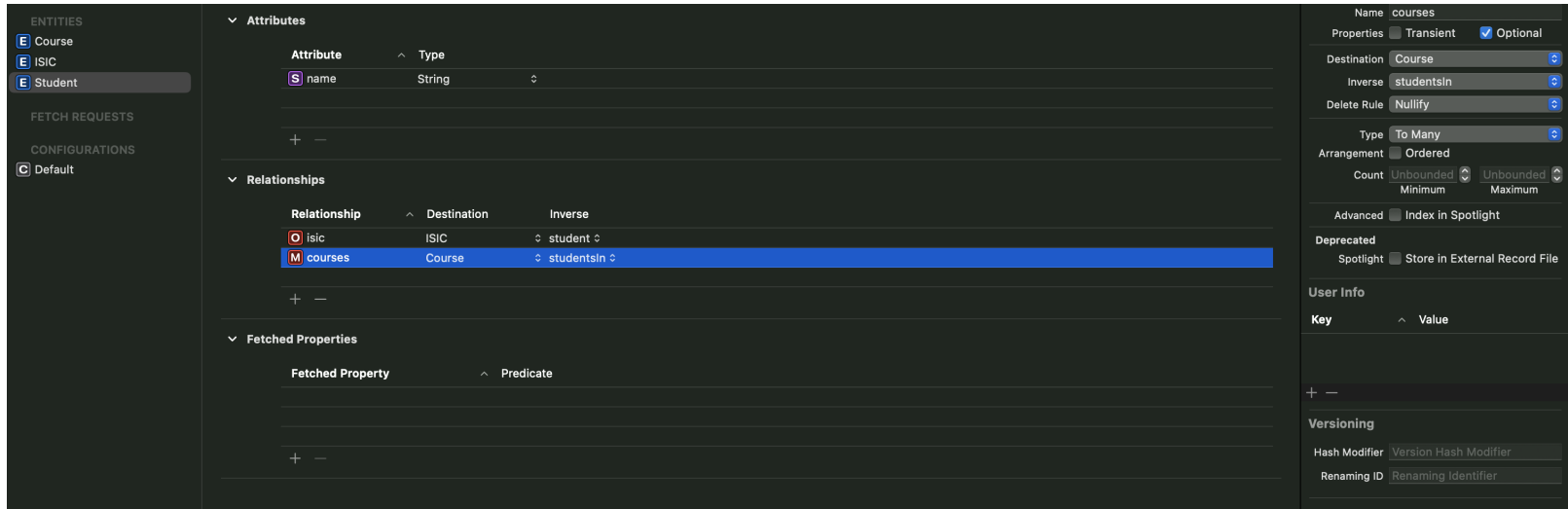
```
@main
struct cd1App: App {
    //
    let persistenceController = PersistenceController.shared

    //
    var body: some Scene {
        WindowGroup {
            ContentView()
            // !!!
            .environment(\.managedObjectContext,
                persistenceController.container.viewContext)
        }
    }
}
```

MOC dostupný v aplikaci:

```
PersistenceController.shared.container.viewContext
```

XCode



Case-study:

- Entity: Student (name), ISIC (idNumber), Course
- relationships

Entity v XCode

Automaticky generované během překladu. Skryté.

```
//
@objc(ISIC) public class ISIC: NSObject {
}

//
extension ISIC {
    //
    @nonobjc public class func fetchRequest() -> NSFetchRequest<ISIC> {
        return NSFetchRequest<ISIC>(entityName: "ISIC")
    }

    // !!!! jsou optional
    @NSManaged public var idNumber: String?
    @NSManaged public var expires: Date?
    @NSManaged public var student: Student?
}

//
extension ISIC : Identifiable {
}
```

Entity v XCode

```
extension Course {
    @NSManaged public var name: String?
    @NSManaged public var studentsIn: NSSet?
}

// MARK: Generated accessors for studentsIn
extension Course {

    @objc(addStudentsInObject:)
    @NSManaged public func addToStudentsIn(_ value: Student)

    @objc(removeStudentsInObject:)
    @NSManaged public func removeFromStudentsIn(_ value: Student)

    @objc(addStudentsIn:)
    @NSManaged public func addToStudentsIn(_ values: NSSet)

    @objc(removeStudentsIn:)
    @NSManaged public func removeFromStudentsIn(_ values: NSSet)
}
```

Co dál?

- Jak se vytvoří objekt a prováží vazby.
- DB dotaz (fetch).
- Napojení výsledků dotazu na UI.
- Mini-aplikace demo/case-study.
- Rušení vazeb a mazání objektů.

Atomické operace nad daty (background task).

Vytvoření MANAGED objektu

```
// Managed Object Context
func MOC() -> NSManagedObjectContext {
    //
    PersistenceController.shared.container.viewContext
}
```

Všechny operace nad Managed Objects v MOC:

- vytvoření, smazání
- dotaz (fetch)

MOC generuje systémové události o změnách MO v rámci MOC.
Na ty budeme velmi brzy napojovat UI.

Vytvoření MANAGED objektu

Je to objekt, který je MANAGED.

```
struct ContentView: View {
    // ... беру si MOC referenci
    @Environment(\.managedObjectContext) private var viewContext

    //
    func newStudent() {
        // instanciacie (generovaný init pro třídu)
        let ns = Student(context: viewContext)
        let nasic = ISIC(context: viewContext)
        // zápis do atributů
        ns.name = "Pepa Vonasek"
        // ...
        nasic.idNumber = "fjffjffjffjff"
        // provázání vazeb 1:1
        ns.isic = nasic
        // Ekvivalent: nasic.student = ns
    }
    // ...
}
```

Extension: přidání funkcionality do MO

```
//  
extension Student {  
    //  
    static func CREATE(inMOC: NSManagedObjectContext,  
                       name: String,  
                       andISIC: ISIC) -> Student  
    {  
        //  
        let ns = Student(context: inMOC)  
  
        //  
        ns.name = name  
        ns.isic = andISIC  
  
        //  
        return ns  
    }  
}
```

Extension: přidání funkcionality do MO

```
extension Student {  
    //  
    var displayName: String {  
        name ?? ""  
    }  
  
    //  
    var loadMyCourses: [Course] {  
        // ...  
    }  
}
```

Zapsání vazeb 1:N

XCodem vygenerované metody add/remove pro vazby.

```
// ...  
let ns = Student(context: viewContext)  
let IZA = Course(context: viewContext)  
let IMS = Course(context: viewContext)  
  
// opět: jsou to ekvivalenty (provedeme jeden z nich)  
IZA.addToStudentsIn(ns)  
ns.addToCourses(IMS)
```

Inverzní vazby:

- Lze napojit/rozvázat z obou směrů.
- Kontrola integrity DB.

Atributy ManagedObject-u

Technická podstata db atributů managed objektů.

```
extension Course {  
    // ...  
    @NSManaged public var name: String?  
}
```

- je to extension => `name` zcela jistě není uložená property
- není uložená property ani property wrapper
- `@NSManaged` zřejmě není konvenční property wrapper.
- (skoro) všechny atributy MO jsou **optional**. Znepokojivé.

Proč jsou ty atributy optional?

Atributy ManagedObject-u

Připomínka KVC z Objective-C:

- na properties lze přistupovat `value(forKey: String)` a `set(value: Any, forKey: String)`.
- v Objective-C se properties označily jako `@dynamic` (je stále ve Swiftu).

`NSManagedObject` přetíží metody `value/setValue` a vykonává DB operaci.

Jako u všeho, DB operace jsou "lazy-evaluated".

Odložený zápis: `try! MOC().saveContext()`.

- Kdy provádět "save" paměti (MOC) do DB souboru?

Atributy ManagedObject-u

```
func connect(student: Student, isic: ISIC) {  
    // setter a getter  
    student.isic = isic  
}
```

Operace `student.name=...` a `... = student.name`

- setter a getter jsou funkce (je to kód) s netriviálním chováním
- životní cyklus MO (vytvoření, načtení, uložení)

Extrémně velký důraz na kontrolu, kterým vláknem operaci provádíme.

Důsledky: MOC **synchronně publikuje** událost o změně dat.

Dotazy (Fetch)

- `FetchRequest<Value>` - metadata, popis dotazu a typ výsledku.
- `fetch` - operace nad MOC, výsledek.
- je `throws` .

Fetch

```
//  
extension Student {  
    //  
    static func LOADALL(inMOC: NSManagedObjectContext) -> [Student] {  
        //  
        let fr = NSFetchRequest<Student>(entityName: Student.entity().name!)  
  
        //  
        guard let _result = try? inMOC.fetch(fr) else {  
            //  
            return []  
        }  
  
        //  
        return _result  
    }  
}
```

Fetch, odložené načtení

Fetch je dotaz do současně:

- Databázového uložště (fyzická DB)
- MOC (hraje roli cache nad fyzickou DB)

Primárně Fetch zajistí vytvoření MO v MOC.

- Obsah (atributy) se do-načtou až při prvním getteru.

```
// Sqlite3: SELECT id from Student;
let _objs = Student.LOADALL(...)
// iteruju pres prázdné schránky MO
for i in _obj {
    // getter, další Sqlite3: SELECT
    print(i.name)
}
```

NSFetchRequest

Další atributy:

- `predicate` - `NSPredicate` .(where ...)
- `sortDescriptors` - `[NSSortDescriptor]` (order by ...)
- `fetchLimit` - (limit ...)

```
NSPredicate("name=%@", someName)
```

```
NSSortDescriptor(key: "name", ascending: true)
```

Napojení na UI

MO jsou objekty. Mají (vypočtené) properties.

- ve SwiftUI řešíme způsob implementace Model/ViewModel.
- `List(seznam) { i in ... }`

Pole MO vzniká jako výsledek dotazu, případně vytvořením objektů.

Není úplně 100% přirozené budovat kolekce managed objektů.

- když je objekt "smazán", on stále tak trochu žije.
- kritická chyba v programu.

Napojení na UI #1

```
struct StudView1: View {
    //
    @State var students: [Student] = []
    @Environment(\.managedObjectContext) private var viewContext

    //
    var body: some View {
        //
        List(students) { i in
            //
            Text(i.name ?? "no-name")
        }.onAppear {
            //
            students = Student.LOADALL(inMOC: viewContext)
        }
    }
}
```

Tabulka je zcela mimo události nad MO.

Napojení na UI #2

```
//
class StudModel: ObservableObject {
    //
    @Published var students: [Student] = []
    //
    init(inMOC: NSManagedObjectContext) {
        //
        students = Student.LOADALL(inMOC: inMOC)
    }
}
//
struct StudView2: View {
    //
    @ObservedObject var model: StudModel

    //
    var body: some View {
        //
        List(model.students) { i in
            //
            Text(i.name ?? "no-name")
        }
    }
}
```

Napojení na UI #3

Klíčový prvek CD: NSFetchedResultsController

```
//
struct StudView3: View {
    //
    @Environment(\.managedObjectContext) private var viewContext
    //
    @FetchRequest(
        sortDescriptors: [NSSortDescriptor(keyPath: \Student.name,
            ascending: true)])
    var students: FetchedResults<Student>
    //
    var body: some View {
        //
        List(students) { i in
            //
            Text(i.name ?? "no-name")
        }
    }
}
```

NSFetchedResultsController (FRC)

Datový (model) řídicí objekt instanciováný s:

- MOC - ve kterém MOC pracuje
- fetch - popis dotazu a jeho výstup (Entity)
 - musí mít !!! `sortDescriptor` , jinak exception
- `delegate: NSFetchedResultsControllerDelegate`

Pak se provede dotaz:

```
try! xy.performFetch()
```

Výsledek: `xy.fetchedObjects` - je pole MO.

Funkcionality FRC abstraktně

`FRC<Entity>` je abstrakcí nad `NSFetchRequest<Entity>`.

- Zapouzdřuje výsledek dotazu. `fetchObjects`:
`[Entity]`
- díky povinnému `sortDescriptor` v dotazu je `fetchObjects` uspořádáno.

Dynamika:

- FRC se nějak dozví o změně objektů v MOC.
- znovu vyhodnotí svůj dotaz -> `fetchObjects2`
- porovná `fetchObjects` a `fetchObjects2`
- delegátovi reportuje změny (add,del,update).

Dynamika MOC & FRC

MOC je objektovou pamětí aplikace. Managed Objekty.

(M)Objekty se do MOC dostávají (MO je prvek MOC):

- dotazem fetch
- vytvořením (v MOC)

(M)Objekty se v rámci MOC aktualizují (setter na atribut).

- MO hlásí události změn do MOC,
- MOC hlásí události do okolního světa (notif. centrum).

Vše synchronně. Setter `st.name="Pepa"` může rozjet kaskádu sync operací.

Dynamika MOC & FRC

Volání setteru `st.name="Pepa"` může způsobit synchronně:

- `st` hlásí svému MOC změnu,
- MOC generuje zprávu o provedených změnách,
- zprávu sync posílá přes Notifikační centrum do aplikace
- odběratelé NOTIFC se dozvídají o změně obsahu MOC
- ...jsou to typicky FRC
- ...pokud se jich změna týká (např dotaz nad `Student`), pak sync volají svého `delegate`
- `delegate` zahájí nějaký svůj proces...

NSFetchedResultsControllerDelegate

Původně určen pro aktualizaci UITableView.

- `controllerWillChangeContent` - otevři tabulku ke změnám.
- `controller:didChangeObject:...` - prováděj jednotlivé aktualizace (insert row, delete row, update row).
- `controllerDidChangeContent` - zavři aktualizací režim.

Dynamika MOC & FRC

- Drobné změny do dat lze dělat "přímo".
- Větší změny (import, přepočtení dat, ...) ve vedlejším MOC.
- Hlavní a vedlejší MOC.

Princip:

- uložení kontextu/změn Main-MOC.
- fork => vedlejší kontext (global thread).
 - fetch dat, provedení výpočtu
 - aktualizace z vedlejšího MOC do hlavního MOC
- uložení kontextu/změn Main-MOC.

Simulovaný MOC & FRC

V aplikacích s objekty (namísto struktur) lze úspěšně simulovat princip FRC.

```
//  
class Record: Identifiable {  
  //  
  let id = UUID()  
  //  
  var name: String = "" {  
    //  
    didSet {  
      NotificationCenter.default.post(name: Notification.recCall,  
                                       object: self)  
    }  
  }  
  //  
  init(name: String) { self.name = name }  
}
```

...

```
class Model: ObservableObject {
  //
  @Published var listof: [Record] = []
  var any: AnyCancellable?
  //
  init() {
    // Knihovna COMBINE
    any = NotificationCenter
      .default
      .publisher(for: Notification.recCall, object: nil)
      .sink { n in
        //
        guard let _n = n.object as? Record else { return }

        //
        if let _i = self.listof.firstIndex(where: { $0 === _n}) {
          //
          self.listof[_i] = _n
        }
      }
  }
}
```

NSPredicate

Klasika z Foundation. Třída starší než my... :)

```
NSPredicate(format: String, ...)
```

- %@ — symbol pro vložení hodnoty objektu z argumentů. * *
- %K — key-path.
- ==, =, <, >, AND, OR, ...
- BEGINSWITH — format: "%@ BEGINSWITH 'Hell'", "Hello"
- CONTAINS[cd] — (c)ase insensitivity, (d)iacritic insensitivity.
- LIKE — *, ?

NSPredicate

NSPredicate, keypaths:

- Uložené atributy.
- Relationships (keyPaths):
 - "isic.idNumber == '123'"
 - "studentsIn.@count > 0" - @count
- Pozn.: compound predicate
- Pozn.: transformace predikátu na SQL select where.

NSSortDescriptor

- Jméno klíče
- ascending: Bool.

```
let sd1 = NSSortDescriptor(name: "name", ascending: true)
// ...
fetch.sortDescriptors = [sd1, sd2, ...]
```

Rušení vazeb, mazání objektů

Nulování vazby (inverse se provede automaticky).

```
var s: Student = ...  
// nuluju vazbu  
s.isic = nil
```

Problém: cílový objekt může zůstat nereferecovaný (memory leak).

```
let isic = s.isic  
// 1:1 vazba s inverzni podobou  
s.isic = nil // isic.student = nil  
// mazani  
MOC().deleteObject(isic)
```

Mazání objektu, Delete rule

A -> B. Chceme smazat objekt A, který se váže na B prostřednictvím relation R (delete rule).

- No Action — referencovaný objekt B nedostává žádnou zprávu. Smysl???
- Nullify (impl.) — provede se nulování inverzní vazby a B zůstává.
- Cascade — maže navázaný objekt B. Dále kaskádově.
- Deny — odmítne akci, tj. A lze smazat pouze pokud nemá vazbu na B. Vazba R zamítne smazání A.

Fetch přes 1:N vazbu

Seznam `Student` => detail `Courses` zapsané.

```
//
@FetchRequest(
    sortDescriptors: [NSSortDescriptor(keyPath: \Student.name,
    ascending: true)])
var students: FetchedResults<Student>
//
var body: some View {
    //
    NavigationView {
        List(students) { i in
            //
            NavigationLink(destination: StudentDetail(student: i)) {
                Text(i.name ?? "no-name")
            }
        }
    }
}
```

detail...

```
struct StudentDetail: View {
    // prázdný fetch nad Course
    @FetchRequest var coursesIn: FetchedResults<Course>

    // ... vstup
    let student: Student

    // ...
    init(student: Student) {
        // ...
        self.student = student
        // specifikace where ...
        let _pred = NSPredicate(format: "studentsIn contains %@", student)
        //
        let _st = [NSSortDescriptor(keyPath: \Course.name, ascending: true)]
        // zápis do property wrapperu
        _coursesIn = FetchRequest<Course>(entity: Course.entity(),
                                           sortDescriptors: _sd,
                                           predicate: _pred)
    }

    // body...
}
```

Operace na pozadí

- Komplikovanější DB operace **je vhodné** provádět v odděleném MOC a pak je atomicky překlopit do hlavního MOC.
- ... jinak MOC bude propagovat každou jednotlivou změnu dál do aplikace
- atomičnost je z pohledu generování události o změně MOC

Operace na pozadí

Z pohledu hlavního MOC se toto odehraje atomicky.

```
func zrusJimISICy() {
    // main-MOC nejprve uložit na disk
    try? MOC().save()
    // persistentContainer.new...
    let _backMOC = CD().newBackgroundContext()
    // v kontextu _backMOC !!!
    let _st = Student.LOADALL(inMOC: _backMOC)
    //
    for i in _st {
        //
        if let _isic = i.isic {
            // !!! Odehrává se zatím virtuálně
            i.isic = nil;
            _backMOC.delete(_isic)
        }
    }
    // až tady se to projektuje do MAIN-MOC
    try? _backMOC.save()
}
```


CoreData v XCode

XCode nám do projektu vygeneruje kód pro zapojení CD.

```
import CoreData
///
struct PersistenceController {
    //
    static let shared = PersistenceController()
    //
    let container: NSPersistentContainer
    //
    init(inMemory: Bool = false) {
        container = NSPersistentContainer(name: "cd1")

        //
        if inMemory {
            container.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }

        //
        container.viewContext.automaticallyMergesChangesFromParent = true
        container.loadPersistentStores(completionHandler: { (storeDescription, error) in
            if let error = error as NSError? {
                fatalError("Unresolved error \(error), \(error.userInfo)")
            }
        })
    }
}
```

Operace na pozadí

Z pohledu hlavního MOC se toto odehraje atomicky.

```
func import(from: [EncodedSomething]) {
    // main-MOC nejprve uložit na disk
    try? MOC().save()

    // toto se spustí ve vedlejší vlákne
    CD().performBackgroundTask { _backMOC in
        //
        for i in from {
            // instancuj Student
            // inicializuj
        }
        // až tady se to projektuje do MAIN-MOC
        try? _backMOC.save()
    }
}
```

Synchronizace napříč zařízeními uživatele

V rámci ekosystému je dnes již standard.

Zvyklost:

- Lokální DB, plně offline funkční. Typicky CoreData.
- Doporučení č. 1: Jednoduché Entity bez vazeb.
- Doporučení č. 2: části, které nejsou podstatné pro synchronizaci, zabalit do BinaryData (dokument).

CoreData+CloudKit. Formulace datového schématu.

CoreData mají implementovanou podporu CloudKit.
(IMHO stále s pochybnostmi)

Závěr & Příště

- CoreData - základní prostředek pro lokální DB
- synchronizace...problém

Příště (hardcore backend...):

- Multi-vláknovost, paralelismus
- async/await volání
- výjimky