Beta 2.0

Full Stack iOS Entwicklung mit Swift

WPF im MIM - WS 17/18

Alexander Dobrynin, M.Sc.

Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

Zusammenfassung und Abgrenzung

Demo
Assignment

UserDefaults

- Persistenter Key-Value Store für simple und geringe Datenmengen
  - Key ist ein beliebiger String
  - Value muss eine sog. Property-List (Bool, Int, Double, Float, String, Date, Data) oder eine gültige Kategorie (Array, Dictionary) dessen sein
- Zugriff über das Singleton UserDefaults.standard innerhalb der lokalen App
- Oder UserDefaults(suiteName:)* für Shared-Defaults zwischen lokaler App und App-Targets (Watch-Extension, Today-Widget)
  *Erfordert das Aktivieren von App-Group in den Capabilities
- Generalisiertes set(_: forKey:) und dediziertes integer(forKey:), string(…) oder auch object(…)

```
UserDefaults.standard.set(100, forKey: "Number") // persist
let int: Int = UserDefaults.standard.integer(forKey: "Number") // get, double, string, …
```

- Die Daten werden nicht unmittelbar gespeichert. UserDefaults entscheidet selbst, wann es ein günstiger Zeitpunkt ist. Das Synchronisieren kann man allerdings einfordern mit UserDefaults.standard.synchronize()

Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

Zusammenfassung und Abgrenzung

Demo
Assignment

Archiving und Filesystem

- Archiving meint das Serialisieren und Deserialisieren von beliebigen Daten (Klassen, Structs, Enums) zu Data? und vice versa
    - Der alte Weg ist es eine Obj-C Klasse (erbt von NSObject) konform zu NSCoding zu machen und jede Property explizit en- und dekodieren. Anschließend kann jedes Objekt vom Typ NSCoding mit NSKeyedArchiver auf Disk geschrieben und mit NSKeyedUnarchiver von Disk gelesen werden
    - Der neue (empfohlene) Weg ist es einen Typen in Swift (Klasse, Struct, Enum) konform zu Codable zu machen und das Schreiben und Lesen über den FileManager abzuwickeln
- In beiden Fällen findet eine Transformation von Model -> Data? und Data? -> Model? statt. Anschließend wird der nicht-menschenlesbare Objekt-Graph in eine Datei geschrieben und vice versa
- Die Datei kann entweder vollständig gelesen oder vollständig geschrieben werden. Beim Lesen wird das gesamte Model (bswp. ein Array mit 300 Elementen) in den Speicher geladen
- Abfragen (Queries) auf Ebene der Objekt-Graphen sind nicht möglich. Hier herrscht das “Alles oder Nichts” Prinzip

Archiving und Filesystem

- iOS verfügt über ein unix-artiges Dateisystem (APFS)
- Jede App hat ausschließlich Zugriff auf seinen Speicherbereich (Sandbox)
- Der Speicherbereich hat unterschiedliche Zugriffspunkte mit unterschiedlichen Rechten
    - Application Directory: Executables, etc… read only
    - Documents Directory: vom User erstellte Daten der App. Sichtbar in Files.app (erfordert Opt-In). Persistent und im iTunes-Backup
    - Application Support Directory: nicht vom User erstellt und nicht für den User sichtbar, allerdings wichtig für die App als solches. Persistent und im iTunes-Backup
    - Cache Directory: z.B. Temporäre Dateien und Bilder. Weder Persistent noch im iTunes-Backup
    - viele mehr, siehe Dokumentation
- Das Dateisystem (gerade APFS) hat sehr viele Features und Möglichkeiten. Über die Veranstaltung hinausgehende Details sind der Dokumentation zu entnehmen

Archiving und Filesystem

- Jeder Zugriffspunkt ist eine URL (z.B. /Documents) und wird über den FileManager angefragt

```
let url = try? FileManager.default.url(
      for: .documentDirectory, // entry point. cachesDirectory, ...
      in: .userDomainMask, // always in iOS
      appropriateFor: nil, // ignore ...
      create: true // create if needed
)
```

- URL strukturieren (Unterverzeichnisse, Dateien)

```
// .../documents/Persons.data
let peronsUrl = base.appendingPathComponent(“Persons”).appendingPathExtension(“data")
```

- Data an URL schreiben oder von URL lesen

```
let data = try? Data(contentsOf: url) // read contents of url
try? data?.write(to: url) // write data to url
```

- Persons.data kann nun als “Dokument der App” in der Files.app auftauchen. Hierfür in der Info.plist den Eintrag “Supports Document Browser” hinzufügen und auf “YES” setzen

- Achtung: der FileManager ist nur innerhalb seiner erstellen Queue Thread-Safe

Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

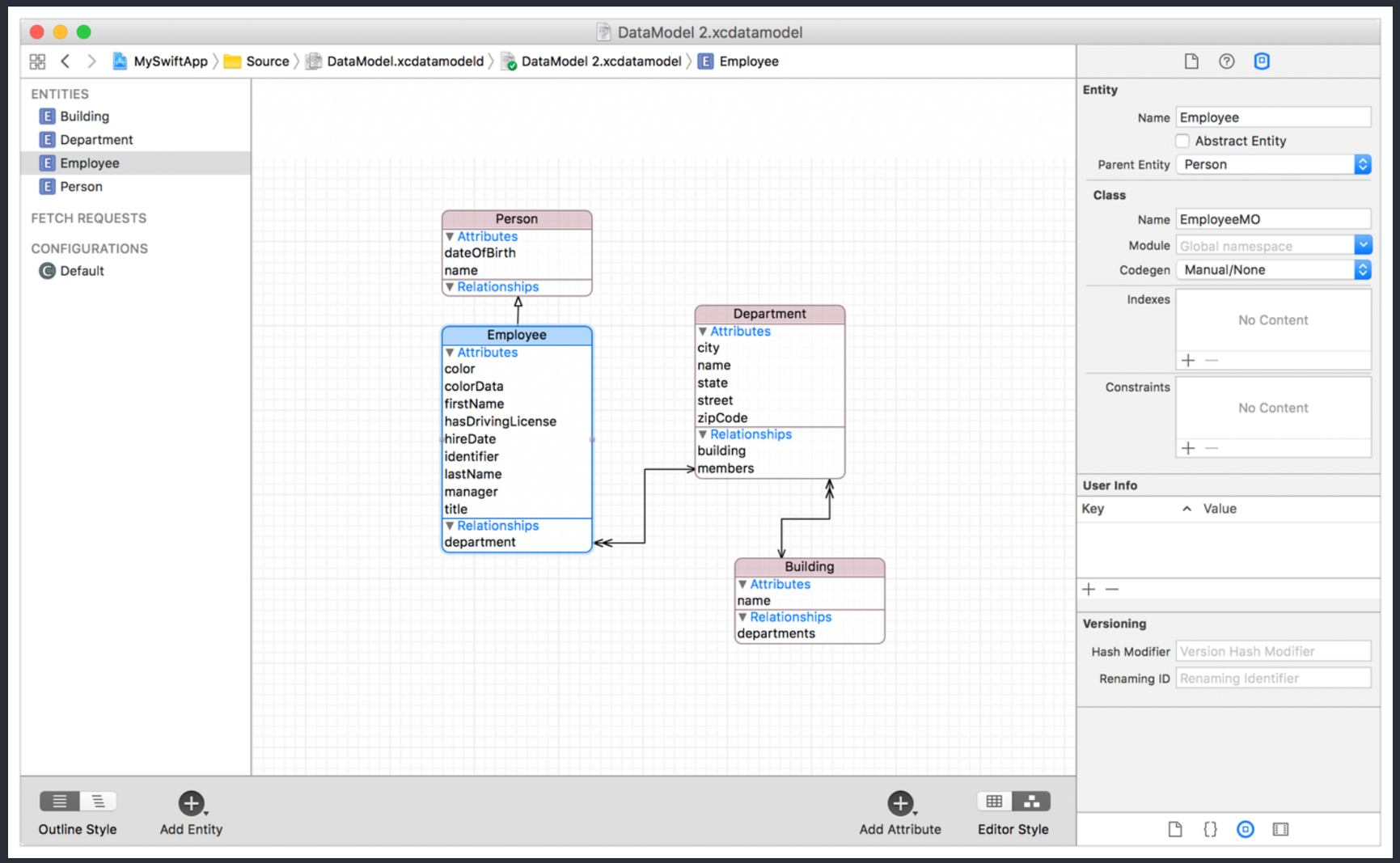
Zusammenfassung und Abgrenzung

Demo
Assignment

SQLite und CoreData

- iOS ermöglicht die Nutzung einer leichtgewichtigen SQL-Datenbank namens SQLite
- SQLite hat eine C-API, weshalb es viele 3rd-Party-Libraries gibt, die einen typisierten Wrapper um SQLite anbieten
- Die häufigste (und von Apple empfohlene) Datenbank ist CoreData, welcher ein Object-Relational-Mapper (ORM) ist und über SQLite mit einer OO-API abstrahiert
- CoreData ist sehr sehr mächtig und damit auch komplex, schwierig zu verstehen und zu debuggen. Des Weiteren passiert sehr viel unter der Haube, was man selbst nicht kontrollieren kann
- Zudem hat die Nutzung von CoreData in den anfänglichen Zeiten von Swift zu vielen Hacks und unelegantem Code geführt, weil CoreData für Objective-C optimiert wurde
- Glücklicherweise wird CoreData nach und nach an die Idiome und Features von Swift angepasst und dadurch immer empfehlenswerter
- Denn CoreData hat äußerst viele Vorteile
    - Entwickelt und selber genutzt von Apple (gegenüber 3rd-Party Libraries)
    - Objektorientierte (ORM) Abstraktion über SQL und dennoch sehr sehr effizient
    - Support für TableViewController (Änderung in der DB -> Änderung in der TableView, animiert)
    - Lazy Dispatching von großen Datenmengen
    - Rollback, Unterstützung von Undo und Redo, Transaktionen, und vieles mehr

SQLite und CoreData



https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/KeyConcepts.html#//apple_ref/doc/uid/TP40001075-CH30-SW1

Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

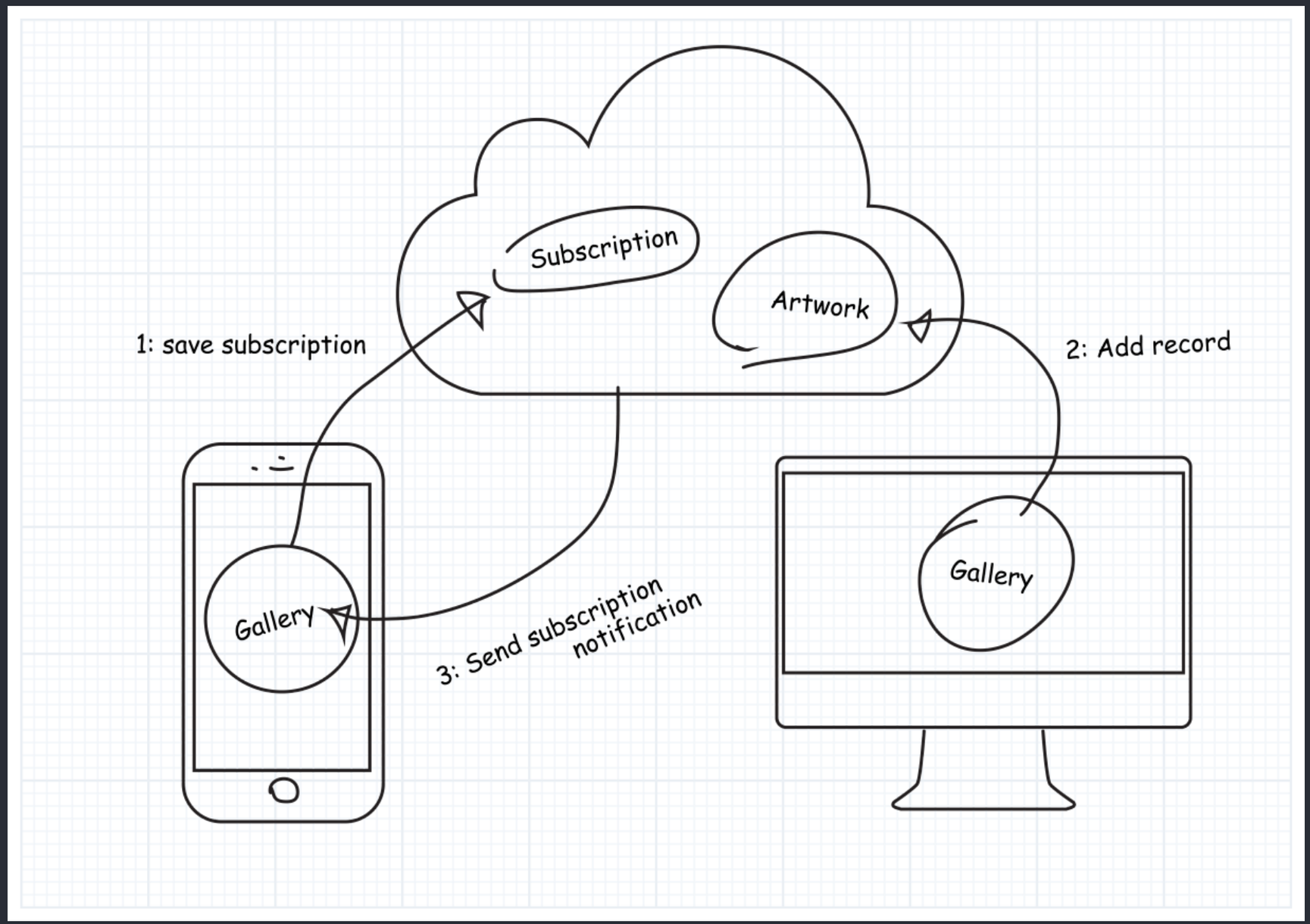
Zusammenfassung und Abgrenzung

Demo
Assignment

CloudKit

- Datenbank in der Cloud*
  Funktioniert nur mit meinem gültigen iCloud-Account
- Dynamisches Schema
- Relationen sind möglich, allerdings nicht so mächtig
- Auch die Abfragen sind nicht so schnell
- Gesamte API ist asynchron, weil jede Abfrage über das Netz geht. Viele Closures und mögliche Errors
- Das wohl wichtigste Feature sind die Subscriptions
  - Device abonniert etwas (Record Type, Prädikat)
  - Jedes mal wenn sich das Abonnement ändert, erhält das Device eine Push-Notification, welche u.a. die Änderung enthält
  - Hinweis: eine Push-Notification ist in erster Linie nur das Übermitteln von Payload zwischen Server und Device (unidirektional). Das Device kann anschließend eine User-Notification präsentieren, muss aber nicht
- Eine Datenbank in der Cloud erscheint seltsam, löst aber viele Probleme (schafft allerdings auch neue)
  - Zentrale und konsistente Daten für Web-, iOS- (iPhone und iPad) und Mac-Apps
  - Migrationen, Aktualisierungen und Versionierungen von Datensätzen und Schemata
  - Backups und Exporte

CloudKit



https://developer.apple.com/library/content/documentation/DataManagement/Conceptual/CloudKitQuickStart/SubscribingtoRecordChanges/SubscribingtoRecordChanges.html

Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

Zusammenfassung und Abgrenzung

Demo
Assignment

Zusammenfassung und Abgrenzung

Tool	Merkmale	Query Support	Aufwand
UserDefaults	Simple Benutzereinstellungen (z.B. Bool, Int, String)	X	kaum
Archiving und Filesystem	Simple und komplexe Objekte. Disk-Caching von Bildern	X	kaum bis gering
SQLite	Relationale Entitäten	✓	hoch
CoreData	Relationale Entitäten mit OO-Semantik und Support für große Datenmengen (TVC)	✓	hoch
CloudKit	Eventgetriebene, konsistente und geteilte Datenbank zwischen verschiedenen Apps mit gleichem Backend	✓	hoch

Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

Zusammenfassung und Abgrenzung

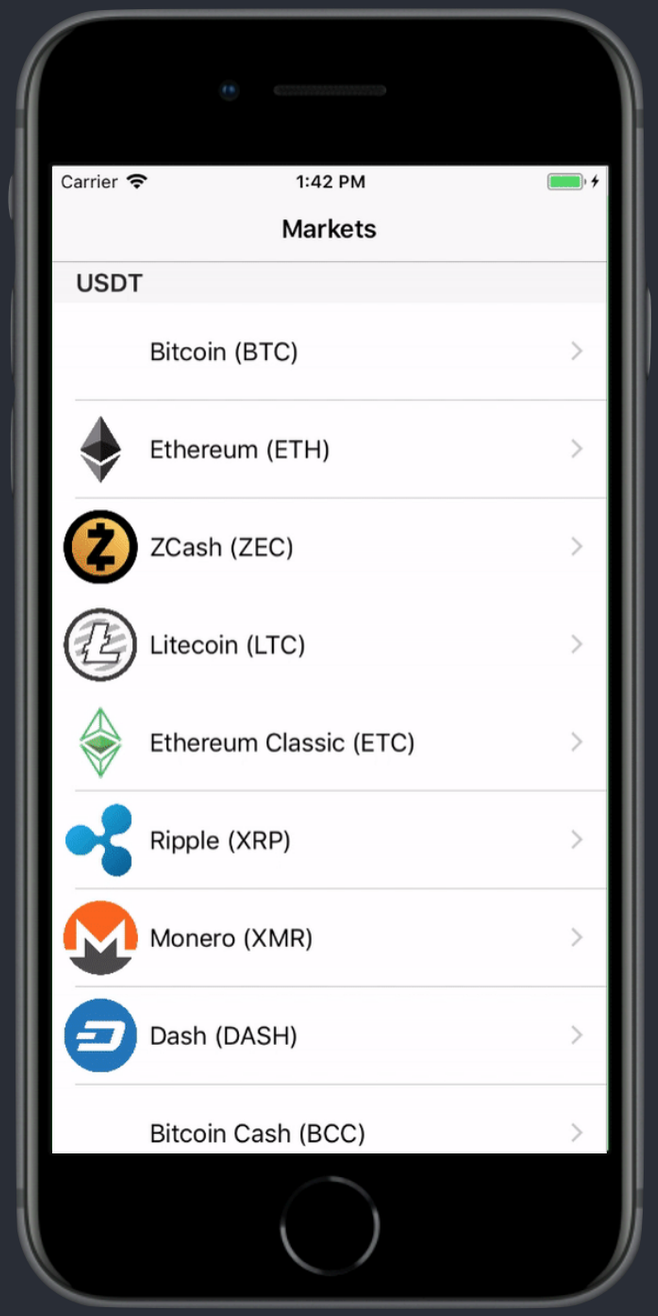
Demo
Assignment

Cryptomarket - Demo

- NSCache und Disk-Storage
- Dokumente in der Files.app

- Generische Protocols mit Type Constraints
- FileManager, URL und Data

- Protocols mit assoziierten Typen
- Pull-to-Refresh in TableViewController



Heute

UserDefaults
Archiving und Filesystem
SQLite und CoreData
CloudKit

Zusammenfassung und Abgrenzung

Demo
Assignment

Cryptomarket - Assignment

- Image Cache persistieren
  - URL für .cachesDirectory einholen
  - Am Besten den Zugriff auf den Cache über
    func get(key:) -> UIImage? und
    func set(img: withData: forKey:) abstrahieren und diese Funktionen anstelle der Cache-Funktionen nutzen
    - Bei get wird erst im Cache geschaut. Wenn es nicht drin ist, dann wird erst auf Disk nachgeschaut
    - Bei set wird erst ins Cache geschrieben und anschließend auf Disk
    - Jedes UIImage soll ich einer dedizierten Datei gespeichert werden. Als Namen kann man url.absoluteString.hashValue verwenden
  - Zwischendurch debugPrint(#function, “from disk/cache”, key) einbauen, um die Funktionsweise zu debuggen
- Sonstige Änderungen und Verbesserungen sind Willkommen
- Bis zum 23.01.18, 13:59 Uhr per Pull-Request einreichen