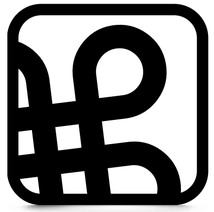


Macoun

Data Model Mal anders

Maxim Zaks
@iceX33

Ablauf

- Philosophie
- Vorstellung von Entitas-Kit
- Ausblick

Worum geht es beim Programmieren?

EDV

=

Elektronische Datenverarbeitung

Zwei Arten von Datenverarbeitung?!

Reine Berechnung

=

Datentransformation/Erzeugung

Reine Berechnung

- Gegebener Datensatz
- Transformation-Regelwerk
- Ergebnis

Compiler
Zustandslose WebServices
Funktionale Programmierung

Was ist mit
Web Browser und Apps?

HTML, CSS & JS rein
Pixelwolke raus

Interaktive Berechnung

Interaktive Berechnung

- Vorheriger Zustand + (User) Eingabe
- Transformation-Regelwerk
- Zwischenergebnis

Die Berechnung wird in einer Schleife durchgeführt

Apps
Spiele
Simulationen

Die Zwischenergebnisse werden in Datenmodell gespeichert

Welche Art von Daten gibt es?

Datenarten

- Applikationsbezogene Daten
- Nutzergenerierte Inhalte
- Laufzeit relevante Daten / Applikationszustand

Applikationszustand mit OOP

Metapher basiert

- Ein Bankkonto
- Eine Person
- Ein Vertrag

Geht es auch mehr Datenorientiert?

Datenorientiert

- DE22100100500123456789 - IBANComponent
- Maxim Zaks - NameComponent
- XC45332FGD - ContractIdComponent

Class
vs.
Entity

Eine gute Klasse

- Versteckt Daten (encapsulation)
- Bündelt verhalten (cohesion)
- Ist SOLID
  - Single responsibility | Open/Closed | Liskov substitution | Interface segregation | Dependency inversion

Eine Entity

- Bündelt beliebige Komponenten
- Hat kein Verhalten
- Ist auffindbar (querable)
- Kann observiert werden

```
let ctx = Context()
let e = ctx.createEntity()
e += NameComponent(value: “Maxim Zaks”)
e += IBANComponent(number: “DE22100100500123456789”)

print(e.get(NameComponent.self)?.value) //Optional(“Maxim Zaks”)

e.with { c: IBANComponent in
 print(c.number) // DE22100100500123456789
}
```

Context
verwaltet
Entities

Komponente ist ein Struct, der `Component` oder `UniqueComponent` implementiert

Entity ist ein Wrapper um
[CID: Component]

Eine Entity ist definiert durch
ihre innere Werte
Nicht durch ihre Geburt

Entities können anhand ihrer Komponenten gefunden werden

```
let group = ctx.all([A.cid, B.cid], any: [C.cid, D.cid], none:[E.cid])

for e in group {
 print(e.has(A.cid))) // true
}

group.withEach { e, c: A in
 print(c)
 e -= A.cid
}
print(group.count) // 0
```

Group ist eine Sequenz von Entity die immer aktuell ist

Implizite und explizite Entity Klassifizierung

```
struct Name: Component { let value: String }
struct BirthDay: Component {let value: Date }
struct NumberOfEmployees: Component {let value: Int}
struct Address: Component {let value: String}

let personMatcher = Matcher(all:[Name.cid, BirthDay.cid])
let organisationMatcher = Matcher(all:[Address.cid], none:[BirthDay.cid])

let people = ctx.group(personMatcher)
let organisations = ctx.group(organisationMatcher)
```

Implizite Klassifizierung durch all, any, none Regeln angewandt an Daten

Explizite Entity Klassifizierung?

```
struct Person: Component {}
struct Organisation: Component {}

let personMatcher = Person.matcher
let organisationMatcher = Organisation.matcher

let people = ctx.group(personMatcher)
let organisations = ctx.group(organisationMatcher)
```

Explizite Entity Klassifizierung durch “TagComponent”

Explizit und Exklusive?

```
struct MyType: Component {
  enum `Type` { case person, organisation }
  let value: Type
}

e += MyType(value: .person)
let group = ctx.group(MyType.matcher)

let people = group.filter { $0.get(MyType.self)?.value == .person }
```

Indexierung der Werte

```
struct Name: Component {let value: String}

let ctx = Context()

let nameIndex = ctx.index { (name: Name) -> String in
    name.value
}

let entities: Set<Entity> = nameIndex["Maxim"]
```

Ziehen vs. Drucken

Context, Group und Entity sind Observable

```
public protocol ContextObserver : Observer {

    public func created(entity: Entity, in context: Context)

    public func created(group: Group, withMatcher matcher: Matcher, in
context: Context)

    public func created<T, C>(index: Index<T, C>, in context: Context)
where T : Hashable, C : Component
}
```

```
public protocol EntityObserver : Observer {

    public func updated(component oldComponent: Component?, with
newComponent: Component, in entity: Entity)

    public func removed(component: Component, from entity: Entity)

    public func destroyed(entity: Entity)
}
```

```
public protocol GroupObserver : Observer {

    public func added(entity: Entity, oldComponent: Component?,
newComponent: Component?, in group: Group)

    public func updated(entity: Entity, oldComponent: Component?,
newComponent: Component?, in group: Group)

    public func removed(entity: Entity, oldComponent: Component?,
newComponent: Component?, in group: Group)
}
```

Applikationszustand kann völlig reaktiv und transparent gemacht werden

Anwendungsmuster

resiapp.io

CollectionView oder TableView Basierend auf einer Group

```
protocol EntityCell {
    func setEntity(_ e : Entity)
}
```

```
let message = messageGroup.sorted()[indexPath.row]
let cellId = getCellId(message: message)

guard
let cell = tableView.dequeueReusableCell(withIdentifier: cellId)  else {
   return UITableViewCell()
}

if let cell = cell as? EntityCell {
   cell.setEntity(e)
}

return cell
```

```
    func setEntity(_ e: Entity) {

        message.text = e.get(TextComponent.self)?.value

        R_Icon.isHidden = !e.has(ShowRComponent.cid)

        bgImage.image = e.has(ShowRComponent.cid) ? UIImage(named:
"bubble_neu") : UIImage(named: "buble2")

        if let cellSize = e.get(CellSizeComponent.self)?.size {
            adjustMessageAndBackground(…)
        }
    }
```

Service Locator
“Dependency Injection für Arme”

```
struct DocumentManagerRef: UniqueComponent {
    let ref: DocumentManager
}
struct ChatViewRef: UniqueComponent {
    weak var ref: ChatView?
}

func setupAppContext() {
 appContext.setUniqueComponent(DocumentManagerRef(ref: DocumentManager()))
}

enum AppContext {
 static var documentManager: DocumentManager? {
  return appContext.uniqueComponent(DocumentManagerRef.self)?.ref
 }
 static var chatView: ChatView? {
  return appContext.uniqueComponent(ChatViewRef.self)?.ref
 }
}
```

Component als Event
Userinteraktion ist Zustandsveränderung

```
struct LoadingComponent : Component {}

struct LoadedComponent : Component {}

struct ReLoadedComponent : Component {}

struct SelectedAnswerComponent : UniqueComponent {
    let index : Int
}
```

Was ist mit Persistieren?

Was ist mit Multi Threading?

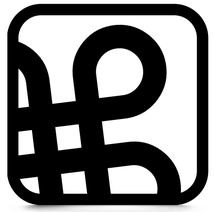
EntitasKit vs. CoreData

- Leichtgewichtig
- Observable
- InMemory (persistieren auf eigene Hand)
- Datenzentriert, kein OOP
- Kein Schema (sehr gut zum improvisieren)

Fragen?

https://github.com/mzaks/EntitasKit

Vielen Dank



Macoun