Swifty Dependency Injection

Enno Welbers (@mkernel)

Dependency Injection?

Als Dependency Injection wird ein Entwurfsmuster bezeichnet, welches die Abhängigkeiten eines Objekts zur Laufzeit reglementiert: Benötigt ein Objekt beispielsweise bei seiner Initialisierung ein anderes Objekt, ist diese Abhängigkeit an einem zentralen Ort hinterlegt – es wird also nicht vom initialisierten Objekt selbst erzeugt.

— *Wikipedia*

Building Blocks

Komponenten

- Kapseln Funktionalitäten.
- Können abhängig von anderen Komponenten sein.
- Werden von anderen Komponenten benötigt.

Building Blocks

Container

- Verwaltet alle verfügbaren Komponenten
- Fungiert als Fabrik für alle Komponenten.
- Löst die Abhängigkeiten zwischen den Komponenten auf.

DI? In Swift?

Wind!

Wind Basics

Protokolle als "Marker" für quasi alles:

- Abhängigkeiten ("Ich brauche Komponente XY")
- Lifecycle-Management
- Standardimplementierungen

Ohne Dependency Injection

```
class Item {
    var Date:Date;
    var From:String;
    var Subject:String;
    var Body:String;
}

class SummaryGenerator {
    lazy var dateFormatter:NSDateFormatter = {...}();

    func generateSummary(for item:Item) -> String {
        return "\(fmt.stringFromDate(item.Date)) (\(item.From)):\(item.Subject)";
    }
}

class MyCell: UITableViewCell {
    lazy var generator = SummaryGenerator();
    func fill(for item:Item) {
        textLabel.text = generator.generateSummary(for:item);
    }
}
```

Mit DI (1/3)

```
protocol NeedsSummaryGeneration { }

protocol SummaryGenerator {
    func generateSummary(for item:Item) -> String;
}

class SummaryGeneratorImpl: SummaryGenerator, Singleton,IndirectResolver,AutomaticDependencyHandling {
    typealias DependencyToken = NeedsSummaryGeneration
    typealias PublicInterface = SummaryGenerator

    var dependencies:[String:[Component]] = [:]
    lazy var dateFormatter:NSDateFormatter = {...}();

    func generateSummary(for item:Item) -> String {
        return "\(fmt.stringFromDate(item.Date)) (\(item.From)):\(item.Subject)";
    }
}
```

Mit DI (2/3)

```
protocol dependsOnMyCell {}

class MyCell: UITableViewCell, ForeignInstantiable,SimpleResolver,AutomaticDependencyHandling {
    typealias DependencyToken = dependsOnMyCell;
    var dependencies:[String:[Component]] = [:]

    lazy var generator:SummaryGenerator! = self.component();

    override func awakeFromNib() {
        super.awakeFromNib()
        self.resolveMe(in: UIApplication.shared.Container!)
    }

    func fill(for item:Item) {

        textLabel.text = generator.generateSummary(for:item);
    }
}
```

Mit DI (3/3)

```
class AppDelegate: UIApplicationDelegate {

    override init() {
        super.init()

        let container = Container();
        SummaryGeneratorImpl.register(in:container);
        MyCell.register(in:container);
        try! container.bootstrap();
        UIApplication.shared.Container=container;
    }
}
```

Was Wind kann

- Lifecycle-Management (Singleton, Instantiable)
- Direktes und Indirektes Auflösen von Abhängigkeiten
- externes Lifecycle-Management (ForeignSingleton,ForeignInstantiable)
- Abhängigkeiten von UIViewController-Klassen aus Storyboards auflösen
- Abhängigkeiten von NSManagedObject Klassen auflösen

Vorteile

- Vereinfachtes Unit-Testing
- Einfacherer Austausch von Komponenten

Nachteile

- Steile Lernkurve
- Die Lesbarkeit des Codes ist in Gefahr

Play with it

https://github.com/palasthotel/wind