Hochschule Bremen

Exposè

Media-science Bsc.

Roman Quistler

May 13, 2019

Contents

1 Introduction

Das Entwickeln von Applikationen ist nicht einfach.

2 Problem and solution

2.1 Problem

Today's applications has been increasing in complexity over the last few years. Notably in the field of frontend development, the amount of functions has increased [1]. The web development has transitioned from server rendered, page-reloading websites, to modern so called single page applications or SPA's. The same applies to the mobile world: social networks, navigation, sharing and editing files together is a commonly demanded by the user. A large part of applications are thus in exchange with APIs, interacting with local or external databases and communicating with the underlying operating system itself (eg the periodic recording of the location via GPS or WiFi). The challenge a developer faces is to come up with a general approach that bridges the gap between what the users sees, the source code and the outside world - mainly: How to structure the source code. Ähnlich wie bei vielen Problemen die in der Softwareentwicklungen bestehen, existiert auch dieses seit geraumer Zeit und betrifft einen Großteil der Entwickler. Es is somit eine immer wiederkehrendes Problem. Daraus ensteht häufig das Bestreben, allgemeingültige Konzepte zu entwickeln, die dieser Problematik entgegenwirken und einen möglichen Lösungsweg aufzeigen. Die hier gesuchten Konzepte werden als sogenannte "Entwurfsmuster" deklariert. Ihr zielt es, den Aufbau bzw. die Architektur von Quellcode so zu gestalten, dass er modular, flexibel und wiederverwendbar ist. Zeitgleich wird der Entwickler dazu gewzungen einem Schema zu folgen und konistent zu arbeiten. Dies fördert die Qualität und Wartbarkeit der Applikation. Über die Zeit haben sich viele Entwurfsmuster für die Strukturierung von Code innerhalb der Präsentationsschicht entwickelt. Dazu gehören unteranderem, nach "Erscheinungsjahr" sortiert: Model-View-Controller (MVC), Model-View-Presenter (MVP) Model-View-ViewModel (MVVM) und - relativ neu im Bunde - Model-View-Intent (MVI). Jedes der hier aufgezählten Muster dient dabei dem gleichen Zweck: Die strikte Trennung der Benutzeroberfläche von der ihr zugrundeliegenden (Geschäfts) Logik. Auch MVI bedient sicher dieser Idee. Brought to life by André (Medeiros) Staltz, MVI encapsulates the interaction between the user and the application as a cycle, where the data flows unidirectionally. It takes its inspiration from two conecepts: The original MVC as introduced by Trygve Reenskaug in 1979 and the often used javascript libraries redux and react. The idea of a cycle is based on the assumption that the output (e.g. a click) from a user resembles the input for the program. The program in turn produces an output which becomes the input for the user. This concept can be embodied as a chain of mathematical functions: f(g(a())) or view(model(intent(event))). When the UI registers an event (e.g. a click) it will be passed to the intent functions as a parameter. The purpose of the intent-function is to translate the user generated event, as in what the user intended to do into something the application can understand and work with. The model function then uses the output as

its input. Its job is it to create a new model without changing the last one. That's what is called immutability. Finally the view function receives the model as its input and takes care of rendering it (a visual representation of the model for the user). In order to achieve the cycle effect and the unidirectional data flow, MVI makes use of reactive programmming and the underlying "Observer Pattern". Given the description of MVI, the questions that arises are: What does an event look like? How is immutability achieved? Since MVI does not have a Controller, Presenter or ViewModel: Where does the logic belong? And how is separation of concerns possible? To summarize, it becomes a question of implementation. What does a developer have to do in order to implement or reap the benefits from this concept of MVI?

why reactive, what does reactive solve for MVI? Streams?

mention non-blocking UI?

talk about the functional part?

side effects?

2.2 Solution

In order to reduce the cognitive burden on the developer when implementing the various components of MVI, the aim is to create a small, but opinionated library. The core of the library will be made up

Um den genannten Problem entgegenzuwirken gilt es, ein Muster zu entwerfen, in welchem der "Zustand" das Hauptaugenmerk bildet. Ein gute Methode um mit Zuständen zu arbeiten, bietet das Konzept der endlichen Automaten. Hierbei wird ein Startzustand festgelegt, sowie eine endliche Anzahl von Zuständen, die im System mit Zustandsübergängen erreicht werden können. Zu beachten ist, dass sich das System immer nur in genau einem Zuständ befinden kann und dieser den SSOT (single source of truth) darstellt. Die wohl wichtigste Eigenschaft eines endlichen Automaten ist jedoch, das bei gleicher Eingabe immer die selbe Ausgabe erfolgt. Kurz um: Der Automat befähigt einen, alle Zustände in dem sich ein System befinden kann konsistent zu beschreiben. Unter der Vielzahl an endlichen Automaten fällt die Wahl auf den sogenannten Mealy-Automat. Bei diesem wird der nächste Zustand auf Basis des derzeitigen Zustands und der erfolgten Eingabe ermittelt. Die Eingabe spiegelt dabei ein Ereigniss wieder, welches durch den Nutzer selbst (und seinen Interaktionen) oder durch Komponenten der Peripherie ausgelöst wurde. Zu nennen wäre bswp. der Klick auf einen Knopf oder die Mitteilung des Betriebssystem, das der Standort sich verändert hat. Die Beschreibung eines solchen endlichen Automaten lässt sich - unteranderem - mit mathematischen Funktionen umsetzen. Für dies und der o.g. Prämisse, bei der die Ausgabe (hier: der Zustand) bei wiederholter, gleicher Eingabe nicht abweichen darf, eignet sich ein Konzept welches auf Namen "Pure Functions" hört. Dieses entspringt dem Programmierparadigma der funktionalen Programmierung und aus eben jener sollen mehrere Konzepte übernommen werden: Unveränderliche Daten Strukturen.
Keine Seiteneffekte.
Kein globalen Variablen.
Pattern Matching + Expressions.
Unidirektional.
Um der hohen interaktivität einer Applikation Herr zu werden, wird eine weiteres Programmierparadigma und Abstraktion herangezogen: Reaktive Programmierung. Sie baut auf der funktionalen Programmierung auf und berücksichtig die oben genannten

Eigenschaften. Des weiteren erleichtert sie das arbeiten und koordinieren von "Threads". Um den Umgang mit einem endlichen Automaten zu vereinfachen und den Quellcode in seiner lesbarkeit zu stärken, wird die Entwicklung einer DSL (Domain Specific Language) in Erwägung gezogen.

3 Specific Tasks

Für die Umsetzung der Bachelor Thesis sind folgende Aufgaben von nöten:

- Recherche: Erhebung und Vergleich von weiterer Literatur, für einen tieferen Einstieg in das Thema.
- Funktionale Anforderungen:
  - Das System soll die Arbeit mit Zuständen auf Basis eines endlichen Automaten vereinfachen.
  - Berechenbar.
  - Reaktiv.
  - Unveränderlich.
  - Unidirektional?.
- Nicht-funktionale Anforderungen:
  - Die Applikation wird in Kotlin entwickelt.
  - Es werden RxJava, RxKotlin und RxAndroid verwendet.
- prototypische Implementierung: Entwicklung einer nativen Android Applikation, in der die ermittelten Anforderung umgesetzt werden.
- Evaluation: Bewertung und Diskussion der Ergebnisse

4 Early Structure

Zusammenfassung/Abstract
Eigenständigkeitserklärung/Eidesstattliche Erklärung

1. Einleitung
   - 1.1. Problemfeld/Motivation
   - 1.2. Ziel(e) der Arbeit
   - 1.3. Aufgabenstellung
   - 1.4. Lösungsansatz
2. Anforderungsanalyse
   - 2.1. Funktionale Anforderungen
   - 2.2. Nicht-funktionale Anforderungen
   - 2.3. Übersicht der Anforderungen/Zusammenfassung
3. Grundlagen und verwandte Arbeiten
   - 3.1. Endliche Automaten
   - 3.2. Bidirektionalität
   - 3.3. Funkionale Programmierung
     - 3.3.1. Pure Functions
     - 3.3.2. Expressions (vs Statements)
     - 3.3.3. Function Composition
     - 3.3.4. Monads
     - 3.3.5. Immutability
     - 3.3.6. Persistent Data Strucures
   - 3.4. Reaktive Programmierung
     - 3.4.1. Observer Pattern
     - 3.4.2. Iterator Pattern
     - 3.4.3. Streams
   - 3.5. Vergleich mit bestehenden Design Patterns
     - 3.5.1. MVVM (Model-View-View Model)
     - 3.5.2. MVI (Model-View-Intent)
     - 3.5.3. Zusammenfassung
   - 3.6. Untersuchung von Bibliotheken
     - 3.6.1. Workflow

5 Timetable

Geplanter Starttermin: 1. Juni 2019
Bearbeitungsdauer: 9 Wochen
Tabelle 1 stellt die geplanten Arbeitspakete und Meilensteine dar:

M1	Offizieler Beginn der Arbeit	01.06.2019
	• Einrichtung der Textverarbeitung • Anforderungsanalyse • Verfassen der Thesis: Kapitel 2 (Anforderungsanalyse)	1.5 Wochen
M2	Abschluss der Analysephase	11.06.2019
	• Recherche • Verfassen der Thesis: Kapitel 3 (Grundlagen	1.5 Wochen
M3	Abschluss der Analysephase	11.06.2019
	• Recherche • Verfassen der Thesis: Kapitel 3 (Grundlagen	1.5 Wochen
My	Erste Fassung vollständige Thesis	11.06.2019
	• Korrekturlesen • Drucken / binden lassen	1 Woche
My	Abgabe der Thesis	11.06.2019

List of Figures

Online References

[1] Kevin Ball. *The increasing nature of frontend complexity*. Jan. 30, 2018. url: https://blog.logrocket.com/the-increasing-nature-of-frontend-complexity-b73c784c09ae (visited on 02/04/2019).

[2] Multiple. *Dont Use Exceptions For Flow Control*. Feb. 18, 2019. url: http://wiki.c2.com/?DontUseExceptionsForFlowControl (visited on 02/18/2019).