Hochschule Bremen

Exposè

Media Computer Science B.Sc.

Roman Quistler

May 21, 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 Wi-Fi).
The challenge a developer faces is to come up with a general approach that bridges the gap between what the users sees and the source code of the application - mainly: How to structure the user facing layer or presentation layer in a layered architecture. [2, 3].
Similar to many problems that exist in software development, this also exists for quite some time and affects a large part of the developers. It is therefore a recurring problem. This often results in the endeavor to develop universally valid concepts that counteract this problem and point out a possible solution. The concepts searched for here are declared as so-called "design patterns [4] Its goal is to design the structure and architecture of source code in such a way that it is modular, flexible and reusable. At the same time, the developer is urged to follow a scheme and work consistently. This promotes quality and maintainability of the application. Over time, many design patterns have evolved for structuring code within the presentation layer. These include, among others, the following, sorted in ascending order by year of publication: Model-View-Controller (MVC - 1979) [5], Model-View-Presenter (MVP - 1990) [6] Model-View-ViewModel (MVVM) [7] und - relativly new to the group - Model-View-Intent (MVI - 2015) [8]. Each of the patterns listed here serves the same purpose: the strict separation of the user interface from the underlying (business) logic.
MVI also makes use of this idea. Brought to life by André (Medeiros) Staltz, MVI encapsulates the interaction between the user and the application as a cycle (as shown in figure 1), where the data flows unidirectionally [9]. It takes its inspiration from two conecepts: The original MVC as introduced by Trygve Reenskaug in 1979 [10] and the often used javascript libraries redux [11] and react [12]. 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

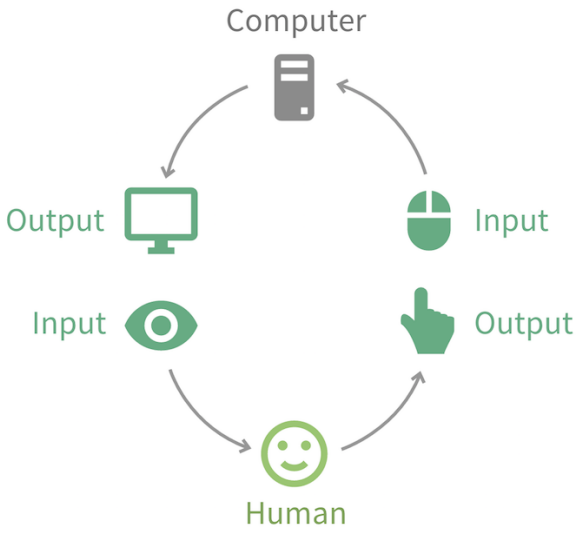

Figure 1: User and Computer as Input and Output

Source: https://cycle.js.org/dialogue.html

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 refered to as immutability. [13] It's a concept that belongs (but is not exclusive) to a programming paradigm called functional programmming [14, 15, 16], where MVI adopts ideas from. 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 [17] and the underlying "Observer" pattern [18].
To sum it up:

> *Model-View-Intent (MVI) is reactive, functional, and follows the core idea in MVC. It is reactive because Intent observes the User, Model observes the Intent, View observes the Model, and the User observes the View. It is functional because each of these components is expressed as a referentially transparent function over streams. It follows the original MVC purpose because View and Intent bridge the gap between the user and the digital model, each in one direction.*
>
> (`cycle.js.org`, Andre Staltz)

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 a "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?

side effects?

state?

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. Hierfür müssen im ersten Schritt die Ereignisse die von einer Aktion des Nutzer ausgehen eindeutig identifiziert und beschrieben werden. Über eine ”intent”-Funktion die bereitgestellt wird, wird das Ereignisse in etwas übersetzt, dass die Intention im Kontext der Applikation deutlich macht. Um die Unterscheidung der Ereignisse übersichtlich zu gestalten, wird das Verfahren des ”Pattern Matching” herangezogen. So kann ein langer Klick auf ein Item innerhalb einer Liste bswp. als die Intention der Löschung interpertiert werden. Im nächsten Schritt erfolgt die Ausführung dieser Intention in der ”model”-Funktion. Sie erhält das derzeitige Ereigniss und Model als Parameter. Zu jeder Intention/jedem Event gesellt sich eine zugehörige Funktion. Auch hierfür gibt die Bibliothek eine Struktur her, durch die diese Art von Mehtoden sichtbar gekennzeichnet werden. Dies führt automatisch zu einer Aufteilung der Logik und erhöht deren lesbarkeit.
Dabei kann es unteranderem vorkommen, dass der Entwickler auf die Geschäftslogik zugreifen muss und somit Seiteneffekte entstehen. Dazu wird seitens der Bibliothek mittels reaktiver Programmierung Abhilfe geschaffen. Zusätzlich wird an diesem Punkt dem Entwickler das Model zur Verfügung gestellt. Es handelt sich dabei um eine unveränderliche Datenstruktur (Objekt), die den Zustand der Applikation beschreibt. Im Falle eines Anmelden Bildschirm wird z.B. hier die vom Nutzer eingetragene Email hinterlegt. Die Unveränderlichkeit macht es unmöglich, das Objekt an andere Stelle ungewollt zu verändern. Darüberhinaus kann auf das Objekt von mehreren ”Threads” gleichzeitig zugergriffen werden und steht somit einer asynchronen Programmierung nicht im Wege. Die ”model”-Funktion erwartet einen neuen Zustand (oder: ein neues Model) als Rückgabe Wert. Aufgrund der Unveränderlichkeit bleibt der vorherige Zustand unangetastet. Stattdessen wird er kopiert und zeitgleich mit neuen Werten versehen. Des weiteren wird der Entwicker dadurch davon abgehalten, das Zustands Objekt als globale Struktur anzulegen. Das Prinzip ähnelt das eines endlichen (Mealy) Automaten und ist deshalb als mögliche Implementation für die ”model”-Funktion innerhalb der Bibliothek angedacht. Für die oben beschriebene Logik wird dem Entwickler ein Klasse, ähnlich dem eines ViewModel oder Presenter an die Hand gegeben. Diese trennt die Benutzeroberfläche (View) von ebendieser Logik. Zuletzt erhalt die View eine einzige Funktion, in der der veränderte Zustand gerendert wird.
Um den Umgang mit der Bibliothek 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:
  –
- Nicht-funktionale Anforderungen:
  –
- Rahmenbedingungen:
  – Entwicklung in Android (Kotlin)
  – Nutzung von RxJava/Kotlin/Android
- 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. MVI
   3.2. Endliche Automaten
   3.3. Unidrektionalität
   3.4. Funkionale Programmierung
      3.4.1. Pure Functions
      3.4.2. Expressions (vs Statements)
      3.4.3. Function Composition
      3.4.4. Monads
      3.4.5. Immutability
   3.5. Reaktive Programmierung
      3.5.1. Observer Pattern
      3.5.2. Iterator Pattern
      3.5.3. Streams
   3.6. Vergleich mit bestehenden Design Patterns
      3.6.1. MVC (Model-View-View Model)
      3.6.2. MVP (Model-View-Intent)
      3.6.3. MVVM (Model-View-Intent)
      3.6.4. Zusammenfassung
   3.7. Untersuchung von bestehenden Bibliotheken

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] guru99. *N Tier(Multi-Tier), 3-Tier, 2-Tier Architecture*. url: https://www.guru99.com/n-tier-architecture-system-concepts-tips.html#3-Tier (visited on 05/17/2019).

[4] TechTerms. *Design Pattern*. July 21, 2016. url: https://techterms.com/definition/design_pattern (visited on 05/16/2019).

[5] Wikipedia. *Model–view–controller*. url: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller (visited on 05/17/2019).

[6] Wikipedia. *Model–view–presenter*. url: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter#History (visited on 05/17/2019).

[7] John Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. url: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/ (visited on 05/17/2019).

[8] Andre Staltz. *What if the user was a function? by Andre Staltz at JSConf Budapest 2015*. Youtube. June 4, 2015. url: https://www.youtube.com/watch?v=1zj7M1LnJV4 (visited on 05/18/2019).

[9] Redux. *https://redux.js.org/basics/data-flow*. Dec. 3, 2018. url: https://reactjs.org/ (visited on 05/20/2019).

[10] Wikipedia. *TrygveReenskaug*. 2000. url: https://en.wikipedia.org/wiki/Trygve_Reenskaug (visited on 05/18/2019).

[11] Dan Abramov and Andrew Clark. *Redux - A predictable state container for JavaScript apps*. url: https://redux.js.org/ (visited on 05/20/2019).

[12] Facebook. *React - A JavaScript library for building user interfaces*. url: https://reactjs.org/ (visited on 05/20/2019).

[14] *Functional programming*. url: https://wiki.haskell.org/Functional_programming (visited on 05/21/2019).

[15] Mary Rose Cook. *A practical introduction to functional programming*. url: https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming (visited on 05/21/2019).

[17] Andre Medeiros (Staltz). *The introduction to Reactive Programming you've been missing*. 2014. url: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754 (visited on 05/20/2019).

[18] Wikipedia. *Observer pattern*. url: https://en.wikipedia.org/wiki/Observer_pattern (visited on 05/20/2019).

Book References

[3] Mark Richards. *Software Architecture Patterns*. O'Reilly Media, Apr. 2, 2015, pp. 1–5.

[13] Joshua Bloch. *Effective Java - Third Edition*. Addison-Wesley Professional, Dec. 17, 2017, pp. 80–86.

[16] Graham Hutton. *Programming in Haskell 2nd Edition*. Cambridge University Press, Sept. 1, 2016, pp. 4–6.