Hochschule Bremen

Exposè - Development of an MVI framework for the Android platform

Media Computer Science B.Sc.

Roman Quistler

May 28, 2019

Contents

1 Introduction

Das Entwickeln von Applikationen ist nicht einfach.

2 Problem and solution

2.1 Problem

Applications today have increased in complexity compared to its counterpart just a few years ago. Especially 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 common want by the user. A large part of applications are thus interacting with APIs, accessing 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 parts of the codebase also known as the presentation layer [2, 3] in a layered architecture. [4].

Like many other problems in software development, this one is a problem that has existed for quite some time and concerns a large part of the developers. It is therefore a common problem. This often results in the endeavour to develop a universally valid concept that counteracts these problems and points out possible solutions. The sought out concept here is defined as an "architectural" pattern. [5, 6, 7] Its goal is to lay out a structure and architecture of source code in such a way that it is modular, flexible and reusable. At the same time, the developer is invited to follow a pattern and produce consistent and readable code. This promotes quality and maintainability of the application. Over time, many "architectural" 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) [8, 9, 10, 11], Model-View-Presenter (MVP - 1996) [12, 13, 14], Model-View-ViewModel (MVVM) [15, 16, 14] und - relatively new to the group - Model-View-Intent (MVI - 2015) [17, 18, 19]. 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 [20, 21, 22]. It takes its inspiration from three concepts: The original MVC as introduced by Trygve Reenskaug in 1979, the JavaScript framework Redux [23] and the JavaScript library React [24]. 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:

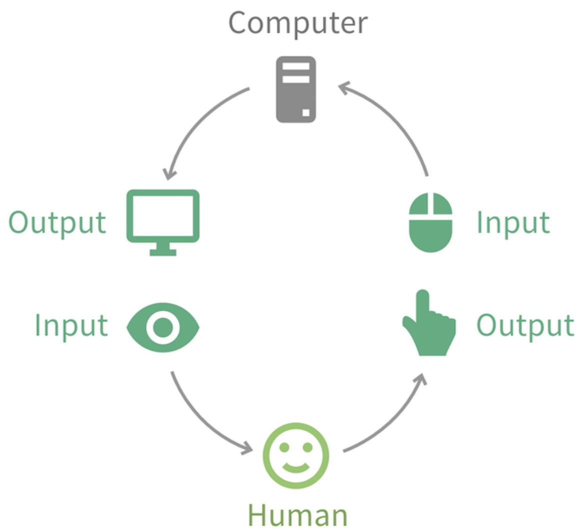

Figure 1: User and Computer as Input and Output

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

f(g(a())) or view(model(intent(event))). When the user interface 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 into something the application can understand and work with. In addition, the event contains data that is necessary for further processing. This requires the event to be accurately described, unique and expressive. The model function then consumes the output (intent function) as its input. Its job is it to create a new model without changing the last one. That's what is referred to as immutability. [25, 26] It's a concept that belongs (but is not exclusive) to a programming paradigm called functional programming [27], where MVI adopts some of its 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 programming [28, 29] and the underlying "Observer" pattern []. To quote Andre Staltz:

> *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.* ([30], Andre Staltz)

Due to MVI being based on functional paradigms and hence using pure functions, it has to fight a common problem in applications called ”side effects”. As mentioned above, it is ofen a requirement to access APIs or a local database, but calling them inside of a pure function is technically not allowed.

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

Given this description of MVI and the mentioned problems, the questions that arise 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 user user-interface logic belong? And how is separation of concerns possible? To summarize, it becomes a question of implementation and design. What does a developer have to do in order to implement or reap the benefits from this concept of MVI?

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 framework. (The goal is to provide the developer with structures, functions and components ...)

At the beginning, the developer is encouraged to define the model. It is suppose to describe the state of the application and is therefore expected to contain variables which characterize this very state.
The framework aims to check whether the model and its variables are immutable to make sure the developer adheres to the principle of immutability, which MVI (and its functional core) demands. This ensures that the developer does not forget it. Due to the immutability the developer must assure that every variable has an initial value. At the end of this process an initial state is created.

The next step will be to uniquely identify and describe the events that are caused by a user interaction. As an example, a long-press on an item within a list can be interpreted as the intention of deletion. For this the framework should provide a structure that the developer uses to mark and visibly identify this kind of events. The events then should be passed as an argument into a predefined ”intent” function, which only takes marked events as a valid input. Thus the developer will be forced to name every event that can possibly happen and to structure the part of the code. The ”intent” function itself can serve as an entrypoint to the unidirectional flow of the framework (and thus MVI). It will be realised through the reactive programming style that the framework api strives to implement.

Based on the type of event the developer might want to execute parts of his business logic. This can lead to mentioned ”side effects” and should be supported by the framework to keep up the unidirectional data flow. The developer only has to tell the framework what logic he intends to call and the task of the framework is then to execute the logic safely and return the result (if any) without the developer having to intervene.

At this point the developer gets access to the model/state structure he has created before. It's the only point in the cycle (or unidirectional flow) where the developer gets hold of his state. This and the immutable property of the state makes it impossible to unintentionally change the object from elsewhere. It leads to code that is more predictable and less prone to errors. In other words: The model/state structure functions as the "single source of truth".

Since the model function expects a new state as a return value, the

In the next step, this intention is executed in the "model" function. It receives the current event and model as parameter. An associated function is added to each intention or event (I didn't get that. Is that part of your solution? If so, which problem does it solve?). The library provides a structure that visibly identifies this type of method for the developer.
This automatically leads to compartmentalization of the logic and increases its readability. Among other things it can happen that the developer has to access the business logic [] and thus side effects [] occur. Since it's enforced on the developer on most mobile platforms to defer heavy operations onto background threads, this must be taken into account. The library can remedy this by using a reactive programming concepts.
In addition, the model is made available to the developer at this point. This is an immutable data structure (object) that describes the state of the application. In the case of a login screen, for example, the e-mail entered by the user is stored here. The immutability makes it impossible to unintentionally change the object from elsewhere. In addition, the object can be accessed by several "threads" at the same time and thus does not stand in the way of asynchronous programming. The "model" function expects (Expects? I don't understand anything right now. How about "generates / creates" or something? The model is transferring the old model into a new one, isn't it?) a new state (or: a new model) as return value. Due to the immutability, the previous state remains untouched. Instead, it is copied and provided with new values at the same time. Furthermore, this prevents the developer from creating the state object as a global structure. It serves as a "single source of truth".
The principle is similar to that of a finite (Mealy) automaton and is therefore conceived as a possible implementation for the "model" function within the library. For the logic described above, a class similar to that of a ViewModel or Presenter is provided to the developer. This separates the user interface (view) from this logic. Finally, the view gets a single function in which the changed state is rendered.
The development of a DSL (Domain Specific Language) (Still very fuzzy, as to what such a DSL could look like and what it brings to the developer.) is being considered in order to simplify the handling of the library and to strengthen the readability of the source code (All in all, it remains unclear to me what the result will be. In the first sentence, you write something about a library, followed by a few more details that don't really sharpen the "big picture". What does a developer gain from your library, to what extent does it make his life easier?).

3 Specific Tasks

The following tasks are necessary for the realization of the Bachelor Thesis:

- Research: Collection and comparison of further literature, for a deeper understanding of the topic.
- Requirement analysis:
  - Small API
  - Inversion of Control
  - Separation of Concerns
  - Immutability
- Design: Detailed design of an architecture for a framework to support the implementation of MVI in mobile (Android) apps.
- Prototypical implementation: Development of a framework in which the determined requirements are implemented. Prototypical use of the framework within an example application.
- Evaluation: Review and discussion of the results.
- Quality assurance: Verifying intended behaviour of the framework via tests.

4 Early Structure

Summary/Abstract
Statutory Affidavit/Declaration of Authorship

1. Introduction
   1.1. Problem area/Motivation
   1.2. Objective(s) of the work
   1.3. Task definition
   1.4. Solution (approach)
2. Requirements analysis
   2.1. Functional requirements
   2.2. Non-Functional requirements
   2.3. Overview of requirements/Summary
3. Fundamentals and related work
   3.1. Flux
      3.1.1. Unidirectional data flow
   3.2. Redux
   3.3. Functional programming
      3.3.1. Pure Functions
      3.3.2. Immutability
      3.3.3. Expressions (vs Statements)
      3.3.4. Function Composition
   3.4. Reactive Programming
      3.4.1. Observer Pattern
      3.4.2. Iterator Pattern
      3.4.3. Streams
   3.5. MVI
      3.5.1. MVC (Model-View-Controller)
      3.5.2. View
      3.5.3. Intent and Action
      3.5.4. Model and State
      3.5.5. Middleware
      3.5.6. Finite state machine

5 Timetable

Planned start date: 15 June 2019
Processing time: 9 weeks
Table 1 shows the planned work packages and milestones:

M1	Official beginning of the work	15.06.2019
	• Setting up the word processor • Requirements analysis • Writing the Thesis: Chapter 2 (Requirements Analysis)	1.5 weeks
M2	completion of the analysis phase	22.06.2019
	• Research • Writing the Thesis: Chapter 3 (Fundamentals)	1.5 weeks
My	First version of the complete Thesis	22.06.2019
	• Proofreading • Printing and binding	1 week
Mx	Submission of the thesis	22.06.2019

List of Figures

Book References

[2] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, Jan. 13, 2013, pp. 19–22.

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

[4] Donald Wolfe. *3-Tier Architecture in ASP.NET with C sharp tutorial.* SitePros2000.com, Jan. 13, 2013.

[5] Mark Richards. *Software Architecture Patterns.* O'Reilly Media, Feb. 2015.

[6] Douglas C. Schmidt Frank Buschmann Kevin Henney. *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages.* John Wiley Sons, Apr. 30, 2007.

[7] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education India, Oct. 2000.

[11] Brad Wilson Jon Galloway, K. Scott Allen, and David Matson. *Professional ASP.NET MVC 5.* Pearson Education India, Aug. 4, 2014.

[14] Adam Freeman and Steven Sanderson. *Pro ASP.NET MVC 3 Framework.* Apress, June 26, 2011, pp. 68–70.

[18] Sergi Mansilla. *Reactive Programming with RxJS: Untangle Your Asynchronous JavaScript Code.* PRAGMATIC BOOKSHELF, Jan. 12, 2016, pp. 112–114.

[20] Adam Boduch. *Flux Architecture.* Packt Publishing, Jan. 30, 2017, pp. 27, 198, 312.

[21] Robin Wieruch. *Taming the State in React: Your journey to master Redux and MobX.* CreateSpace Independent Publishing Platform, June 5, 2018, p. 14.

[22] Ilya Gelman and Boris Dinkevich. *The Complete Redux Book.* Leanpub, Jan. 30, 2017, pp. 6–7.

[23] James Lee, Tao Wei, and Suresh Kumar Mukhiya. *Redux Quick Start Guide: A beginner's guide to managing app state with Redux.* Packt Publishing, Feb. 28, 2019.

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

[27] Pierre-Yves Saumont. *Functional Programming in Java: How functional techniques improve your Java programs.* Manning Publications, Jan. 27, 2017.

[28] Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava.* O'Reilly Media, Oct. 27, 2016.

Report References

[8] Trygve Reenskaug. *THING-MODEL-VIEW-EDITOR - an Example from a planningsystem*. The original MVC report. May 12, 1979. url: http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf (visited on 05/23/2019).

[9] Trygve Reenskaug. *MODELS - VIEWS - CONTROLLERS*. Dec. 10, 1979. url: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf (visited on 05/23/2019).

[10] Glenn E. Krasner and Stephen T. Pope. *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. ParcPlace Systems, Inc, 1988.

[12] Mike Potel. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java*. Taligent, Inc., 1996. url: http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[16] Erik Sorensen and Marius Iulian Mihailes. *Model-View-ViewModel(MVVM)Design Pattern using Windows Presentation Foundation (WPF) Technology*. University of Southern Denmark, The Maersk Mc-Kinney Moller Institute, Titu Maiorescu University, Faculty Science, and Information Technology, 2010. url: http://megabyte.utm.ro/articole/2010/info/sem1/InfoStraini_Pdf/1.pdf.

[26] Christian Haack et al. *Immutable Objects in Java*. Technical. Apr. 28, 2006.

Thesis References

[24] Naimul Islam Naim. "ReactJS: An Open Source JavaScript Library for Front-end Developement". Bachelor. Metropolia University of Applied Sciences, May 30, 2017.

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).

[13] Martin Fowler. *Model-View-Presenter (MVP)*. July 18, 2006. url: https://martinfowler.com/eaaDev/uiArchs.html#Model-view-presentermvp (visited on 05/23/2019).

[15] John Gossman. *Introduction to Model/View/ViewModel pattern for building WPF apps*. Oct. 8, 2005. url: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/ (visited on 05/17/2019).

[17] 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).

[19] Hannes Dorfmann. *Model-View-Intent on Android*. Mar. 4, 2016. url: http://hannesdorfmann.com/android/model-view-intent (visited on 05/23/2019).

[29] 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).

[30] Andre Staltz. *What MVC is really about*. url: https://cycle.js.org/model-view-intent.html#model-view-intent-what-mvc-is-really-about (visited on 05/23/2019).