Hochschule Bremen

Exposè

Media-science Bsc.

Roman Quistler

March 25, 2019

Contents

1 Introduction

2 Problem and solution

2.1 Problem

Today's applications seem to be increasing in complexity over the last few years. Especially in the field of frontend development, the range 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 user

But the same applies to the mobile world. Social networks, navigation, sharing and editing files together. The functionality that a application includes today

More functionality does thus in some sense translate into more conditions the application can be in.

Considering a simple login screen feature inside an android application. A user can provide authentication credentials in order to login. It's possible that a user enters wrong credentials, which in turn can be handled by the app by displaying a short description of what went wrong, like an error message. If the user then provides the wrong credentials more than three times, his user account could get locked. This also in some way must be expressed by the application. He could also make use of the "Forgot Password" functionality. That could allow the user to navigate away from the application into a browser window instead in order to reset the password.

| State/Event | Logging in | Logging out | | Logged in | x | x | | Logged out | x | x |

Immediately it becomes clear that there are already four possible states in which the application can be in. A developer thus ends up with states * events possible states which must be considered and handled appropriately.

For example the application might be in the state of "logging in" and then a user tries to transition into the event of "logging in" again, which might not be desirable.

The complexity for such a "simple" or at least standard feature grows linearly with the amount of states and events that are introduced into the application.

With the increasing states and events it becomes harder and harder to have a general overview of all possible states.

Given that a project grows and more and more functionality is added to the codebase, the more important the state becomes and the harder it becomes to introduce state as an afterthought.

In most cases **??**, an application communicates with other interfaces, such as a backend or local database. On mobile devices it's also often necessary to know the GPS status, whether mobile data is enabled or not, etc.

A click of a user thus can cause many changes inside and outside of the application. A developer must be aware of all the possible outcomes, react to them and save what has happened.

By increased application complexity the necessity to make sure the application behaves exactly as expected in all cases increases. This property in software development is referred to a deterministic system, which states **??** that a deterministic model always

produce the same output from a given starting condition or initial state. This is a desirable property because it makes it simpler to predict and reason about events and states in which the application can be within.

The problem of opposite, nondeterministic software is thus that it makes the code less readable since it's impractical to gloss over the code to determine the potential output of a function call.

Code readability becomes a vital part in codebases when they start growing in complexity and size. But also when they are being worked on by more people than just the person who initially wrote it.

productive in writing code instead of deciphering other peoples codes it becomes essential.

A product of readability is another important property which is increased maintainability. Maintainability ties into the notion of not having to rewrite parts of the codebase that is not directly related to the functionality being added or modified.

The arguably most well known remedy for this problem is the SOLID principle, first proposed by **??** in his paper **??** published in the year 2000.

* Single responsibility principle * Open-closed principle * Liskov substitution principle * Interface segregation principle * Dependency inversion principle

Thus far it can be summaries the problem to be that of maintainability, flexibility and readability. These are basic concepts that unless respected have a domino-like side-effects

And core to this problem is managing the applications state in a way that it can survive platform restrictions like for example unexpected process death or in android specifically a configuration change ie device rotation.

Once this basic login feature is implemented another problem emerges which is to add more functionality like for example a 'protected view' which requires the user to be logged in this view should not be reachable from the 'logged out' state without transition into 'logged in' state first.

Side-problems solved by solution to main problem

//begin rm In the problem description it has been discussed a myriad of principles and design patterns to allow developers to reach maintainable, flexible and readable code in the context of the Android framework.

But this becomes hard because there are so many, and context matters when it could be respected or ignored.

These patterns and principles can sometimes not make sense in certain scenarios.

Any pattern is only as good as how consistently it is implemented.

To try to help with this problem a helper library can be developed to remove the mundane and boring parts of this approach and let the developer focus on the application flow rather than implementation details needed just to achieve said desired flow.

Since there are several ways to achieve this using different approaches this helper library will be opinionated. //endrm

* Nondeterministic * Closed for extension, open for modification * Unreadable code * Unmaintainable code

The developer must thus manage the state. The problem is to find a good design pattern, which solves the state problem by not treating it as an afterthought.

shared mutable state = bad

in such a way that it respects SOLID/DRY etc

2.2 Solution

mention reactive programming

In order to reduce the cognitive burden on the developer of managing the different conditions and states the application can end up in, a small and lightweight (set of tools), yet opinionated library will be developed. At it's core sits a 'Mealy' state machine. Its output is determined by both its current state and inputs which in this case will be the current events.

It must be deterministic and follow the principles of an unidirectional data flow. The implementation will make use of or try to follow functional concepts. Pure functions (no side effects) will be used whenever it's possible. The exception handling will be facilitated by the use of the 'Either' monad.

The state of the application should be observable and the library should provide a solution for asynchronous actions (side effects) out of the box. The developer should work on streams, rather ...

https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Declarative Imperative

Due to the reasons discussed in the problem description, modern applications has so many real-time UI events that there is really no other way to do it than reactively. Every single event might need to be immediately reflected into a type of persistant storage like a local or remote database. As said by Andreas Staltz **??** on reactive programming:

"Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details."

Reactiveness also introduces the concept of streams or data streams, officially named 'Observable's in Rx libraries. On those we can apply operations like filter, sort or similiar.

Reactivity gives us flexability, loosely coupled and scalable and it also makes it easier to deal with failures and they can also be handled gracefully.

Reactiveness gives responsivness something that is of up most importance for a UI, even if a failure occurs.

The reactiveness also provides the concept of 'dont call us, we call you' principle where streams or observables (as in source) will notify or emit data to the observer.

This is a huge improvement compared to the counter which is polling-based approaches as in asking the source every so often if changes occurred.

1. Core - Finite State Machine (Mealy) 2. Functional programming 3. Reactive 4. Copy-on-write / immutability A technique that blends the advantages of mutable and immutable objects 5.

Moore machine only cares about its current state to achieve an output Mealy machine cares about its current state and the current inputs like events to reach a new output.

1. State

2. Events
3. S.O.L.I.D
4. Functional
5. Reactive
6. Clean code
7. DRY
8. MVI

Since user interactions are inherently asynchronous, in that a user clicks on the screen which trigger some action. That being accessing a remote or local data source, or simply toggle between dark and light themes, one could leverage the concepts of reactive programming. To address the race-condition prone shared mutable state that plagues concurrent/reactive programming we want to use immutable state.

Functional programming provides conecpts such as monads, pure functions (no side effects), and immutability and so referential transparenc. High-Order functions, declaritive.

To be able to know which events and state the developer has to handle, a finite state machine [2] fits well into this problem mainly because its possible to create a deterministic state machine.

By writing a 'state transition table' the developer immediately knows all possible events and states that needs to be handled. This is useful because it makes it easier to plan the architecture of the application.

If we do not address mutable state while doing concurrent computing the application becomes non-deterministic which is bad since it will lead to unexpected behaivor and is defacto broken by design. This is why the state must be immutable in a concurrent context.

This will be solved by writing a sort of helper library, which rather than advocating strict software design patterns guides the user into a type of workflow that is approporiate for a reactive environment with the concepts of Events, State and Side effects between those.

1. code generation for shared.ofType
2. annotation @State, @Event, @SideEffect
3. Reactive kotlin helper library for finite state machine/workflow
4.

In the problem description it has been discussed a myriad of principles and design patterns to allow developers to reach maintainable, flexible and readable code in the context of the Android framework.

But this becomes hard since there are so many, and context matters when it could be respected or ignored.

These patterns and principles can sometimes not make sense in certain scenarios.

Any pattern is only as good as how consistently it is implemented.

To try to help with this problem a helper library can be developed to remove the mundane and boring parts of this approach and let the developer focus on the application flow rather than implementation details needed just to achieve said desired flow.

Since there are several ways to achieve this using different approaches this helper library will be opinionated.

It is essentially a library that facilitates a statemachine, but unlike most state machines it also encourages a certain type of workflow. It will thus be a opinionated library.

List of Figures

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] Wikipedia. *Finite-state machine*. Feb. 18, 2019. url: https://en.wikipedia.org/wiki/Finite-state_machine (visited on 02/18/2019).