REFACTORING FOR

RXJAVA

UBIRATAN SOARES

SEPTEMBER / 2016

MOTIVAÇÕES

RxJava é um dos trending topics no desenvolvimento para Android nos últimos 2+ anos

Você certamente já viu alguma solução “Rxfyed" para algum problema na sua timeline.

Fato : programação reativa oferece soluções poderosas para problemas difíceis

Fato : RxJava irá alcançar o release 2.0.0 em breve, uma atualização significativa com novas funcionalidades e várias mudanças

Seu me projeto não utiliza nada de RxJava hoje, como eu o refatoro para ter acesso à essas benesses divinas?

ESSA PARECE SER UMA ÓTIMA PERGUNTA !

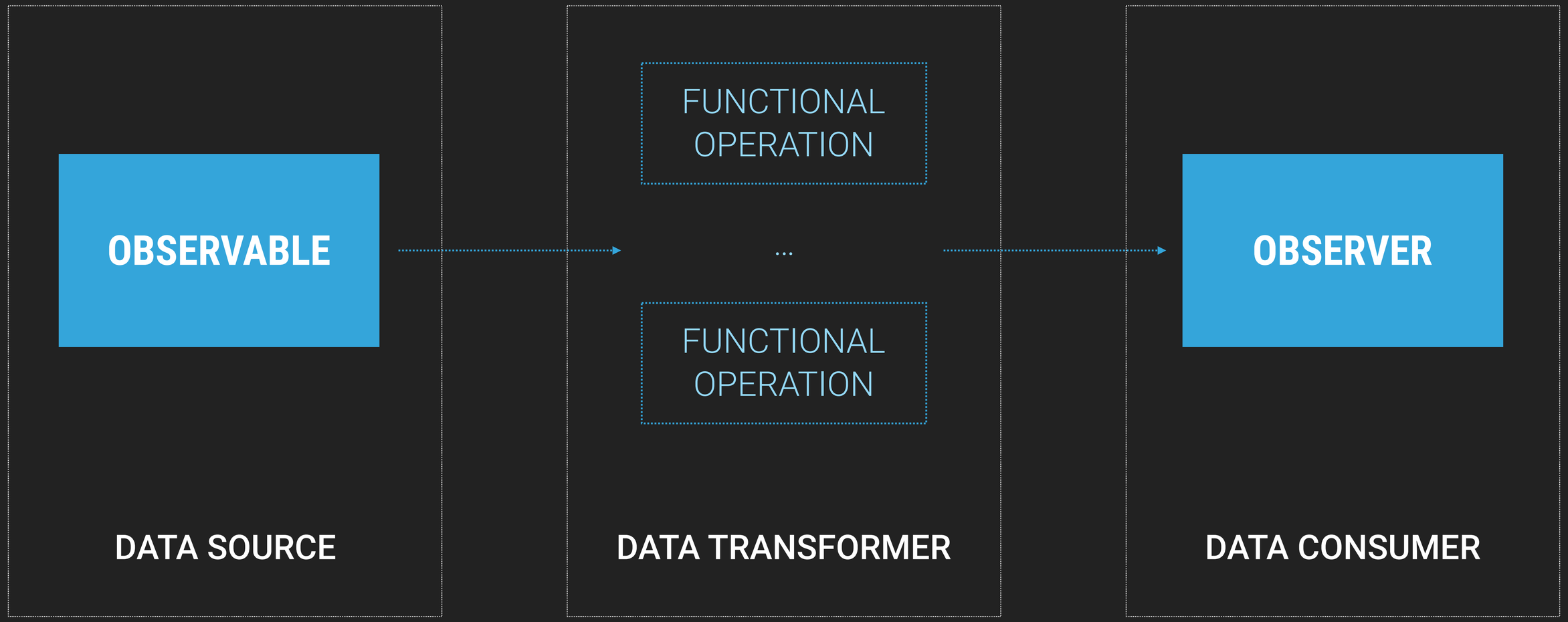
SUA REAÇÃO ?

```
restAPI.endpoint()
        .compose(Transformers::handleNetworkingError)
        .onErrorResumeNext(t-> handleError(t))
        .map(payload -> payload.array)
        .flatMap(Observable::from)
        .filter(DataValidation::validate)
        .map(ModelTransformer::toUI)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribeOn(Schedulers.io())
        .subscribe(
                data -> updateUI(data),
                this::reportError,
                () -> Timber.d("DONE")
        );
```

ANTES DE TUDO

ENTENDER RXJAVA

VISÃO SIMPLIFICADA



UM GUIA ENVIESADO

1. Identifique uma fonte de emissões reativas e defina o tipo de fluxo desses dados
2. Identifique em que ponto da sua aplicação você quer receber esses dados (Observer / Subscriber)
3. Adaptar e evoluir as camadas da sua aplicação para orquestrar o fluxo de dados, ligando fonte a consumidor
4. Se as fontes de dados mais óbvias já esgotaram, hora de avançar para as não-óbvias. Retornar para passo 01

REACTIVE DATA SOURCES

“Like bugs, you can find them everywhere in your code”

- Soares, U.

ANTES (ASYNCTASK)

```
private void notSoTastyThreading(String input) {

    new AsyncTask<String, Void, String>() {
        @Override protected void onPreExecute() {
            notifyProcessingBeforeInit();
        }
        @Override protected String doInBackground(String... params) {
            return processing(params[0]);
        }
        @Override protected void onPostExecute(String result) {
            handleResult(result);
        }
    }.execute(input);
}
```

DEPOIS (THREADING COM RXJAVA)

```
private void beatifulThreading(String input) {

    Observable.just(input)
            .doOnSubscribe(this::notifyProcessingBeforeInit)
            .map(this::processing)
            .subscribeOn(Schedulers.computation())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(this::handleResult);
}
```

ANTES (TIMERTASK)

```
Handler toMainThread = new Handler(Looper.getMainLooper());

TimerTask periodic = new TimerTask() {
    @Override public void run() {
        toMainThread.post(() -> updateUI());
    }
};

Timer timer = new Timer();
timer.schedule(periodic, NO_DELAY, PERIOD_IN_MILIS);


. . .


timer.purge();
```

DEPOIS (TIMER COM RXJAVA)

```
Subscription timer =
      Observable.timer(PERIOD_IN_SECONDS, TimeUnit.SECONDS)
              .observeOn(AndroidSchedulers.mainThread())
              .subscribe(instant -> updateUI());

. . .

timer.unsubscribe()
```

ANTES (RETROFIT VIA CALL PATTERN)

```
api.movieWithId(movieId);
   .enqueue(new Callback<Movie>() {
        @Override public void onResponse(
               Call<Movie> call, Response<Movie> response) {
            if(response.isSuccessful()) {
                // Success
            } else {
                // 4xx or 5xx
            }
        }
        @Override public void onFailure(Call<Movie> call, Throwable t) {
            // Deu ruim mesmo
        }
 });
```

DEPOIS (RETROFIT COM RXJAVA)

```
starWarsAPI.people()
        .subscribeOn(Schedulers.io())
        .flatMap(payload -> Observable.from(payload.results))
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
                data -> addToList(data),
                Throwable::printStackTrace,
                () -> adapter.notifyDataSetChanged()
        );
```

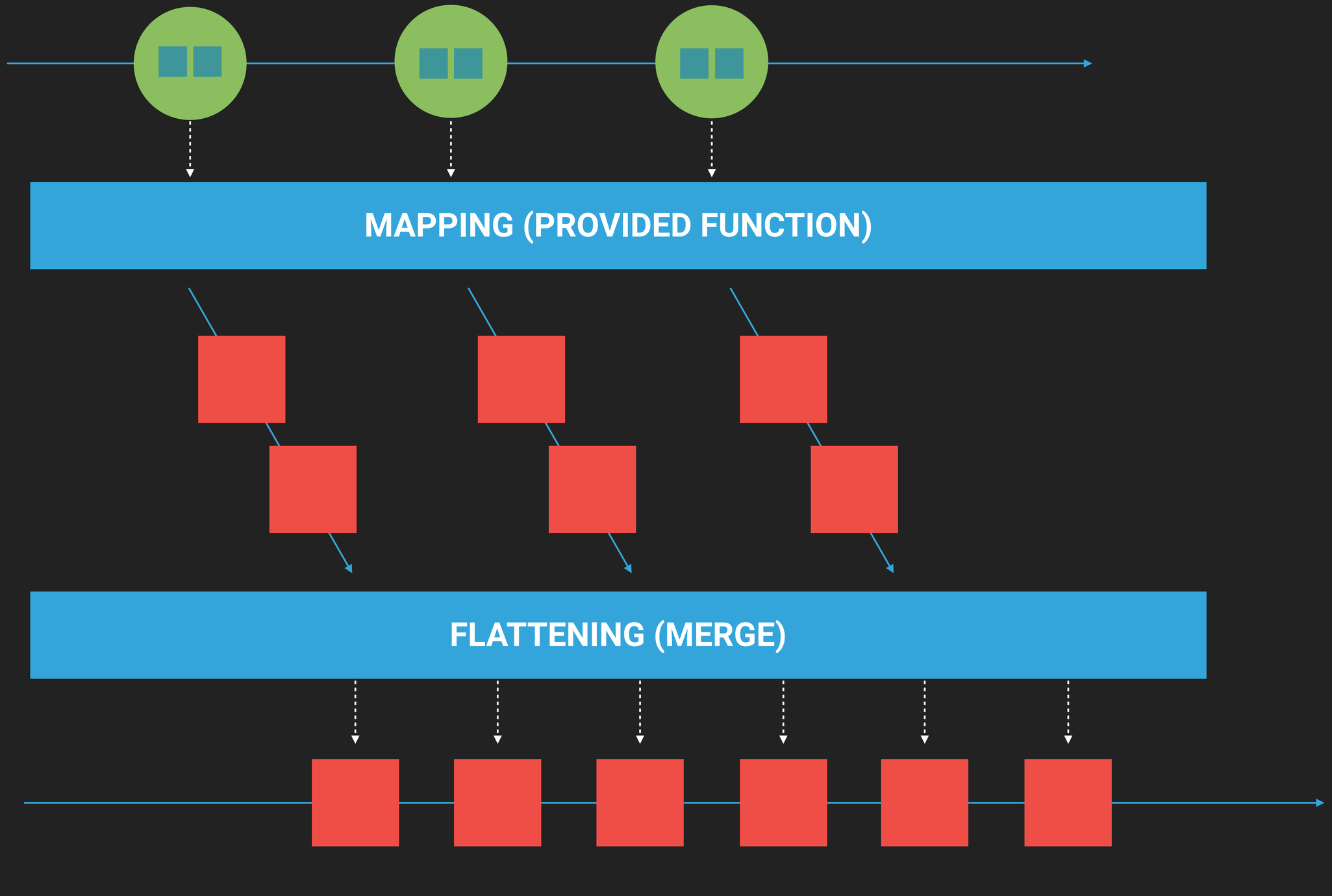
E PARA ENCADEAR DUAS OPERAÇÕES ASSÍNCRONAS ?

ANTES : CHAINING CALLBACKS

DEPOIS : CHAINING COM RXJAVA

```
starWarsApi.people()
        .subscribeOn(Schedulers.io())
        .flatMap(payload -> selectRandomPeople(payload.results))
        .doOnNext(System.out::println)
        .flatMap(people -> Observable.from(people.films))
        .flatMap(filmUrl -> {
            String filmId = ResourceIdExtractor.idFromUrl(filmUrl);
            return api.movieById(filmId)
        })
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
                data -> addToMoviesList(data),
                Throwable::printStackTrace,
                () -> moviesAdapter.notifyDataSetChanged()
         );
```

FLATMAP



REACTIVE SOURCES NA INTERFACE

RxBinding to the rescue!

É possível adaptar outros callbacks utilizando Subjects ou o utilitário fromAsync / fromEmitter

Atenção ao lidar com operadores que envolvam tempo : eles já trocam o Scheduler da sequência, é preciso ressincronizar com a UI Thread para atualizações na UI

SEARCHVIEW (RX WAY)

```
@Override public boolean onCreateOptionsMenu(Menu menu) {

    getMenuInflater().inflate(R.menu.menu_search_by_terms, menu);
    MenuItem search = menu.findItem(R.id.search);
    SearchView searchView =
                (SearchView) MenuItemCompat.getActionView(search);

    RxSearchView.queryTextChangeEvents(searchView)
            .debounce(300, TimeUnit.MILLISECONDS)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(this::proceedWithQuery);

    return true;
}
```

DEFINING YOUR OBSERVERS

“Ideas are bullet proof”

- V

OBSERVER (NORMAL WAY)

```
api.getAvaliableItems(),
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Observer<List<Item>>() {
                @Override public void onCompleted() { … }

                @Override public void onError(Throwable e) { … }

                @Override public void onNext(List<Item> items) { … }
         }
);
```

OBSERVER (ACTIONS WAY)

```
Observable.fromCallable(() -> “Better subscribing with actions”)
          .subscribe(
                    System.out::println,
                    throwable -> someErrorAction(),
                    () -> done()
          );
```

IMPORTANTE

SEMPRE IMPLEMENTE onError( ) em seus Observers/ Subscribers, com Actions ou não

Observers / Subscribers causam memory leaks se retidos em Activities / Fragments / etc

Controle no ciclo de vida via Subscription

Evite Subscribers (statefull)

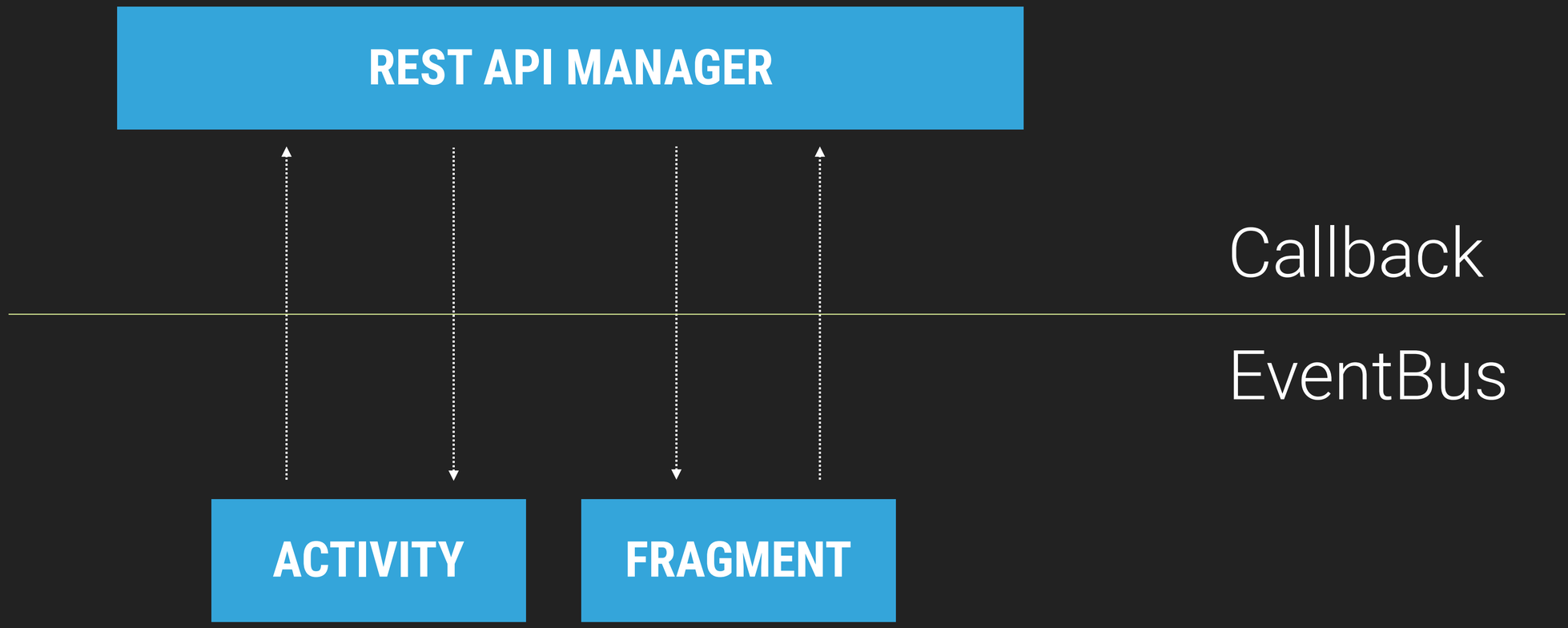
```
Subscription first = Observable.interval(1, TimeUnit.SECONDS)
        .subscribe(System.out::print);

Subscription second = Observable.range(1, 100000)
        .subscribe(System.out::print);

CompositeSubscription subs = new CompositeSubscription();
subs.add(first);
subs.add(second);


// . . .

first.unsubscribe();

subs.add(third);

if(subs.hasSubscriptions()) subs.unsubscribe();
```

EVOLVING YOUR ARCHITECTURE

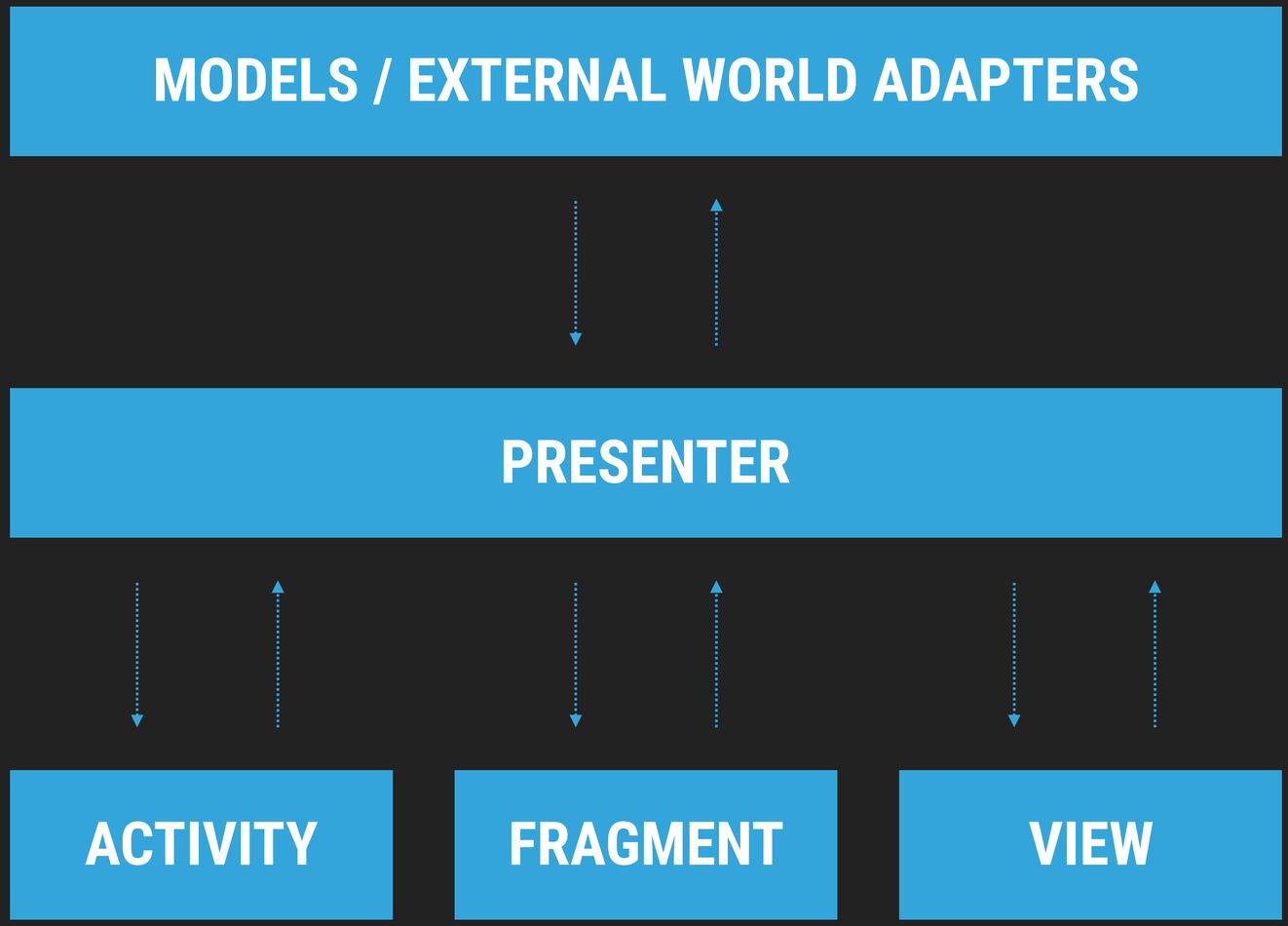
“Tradeoffs? Wellcome to Engineering”

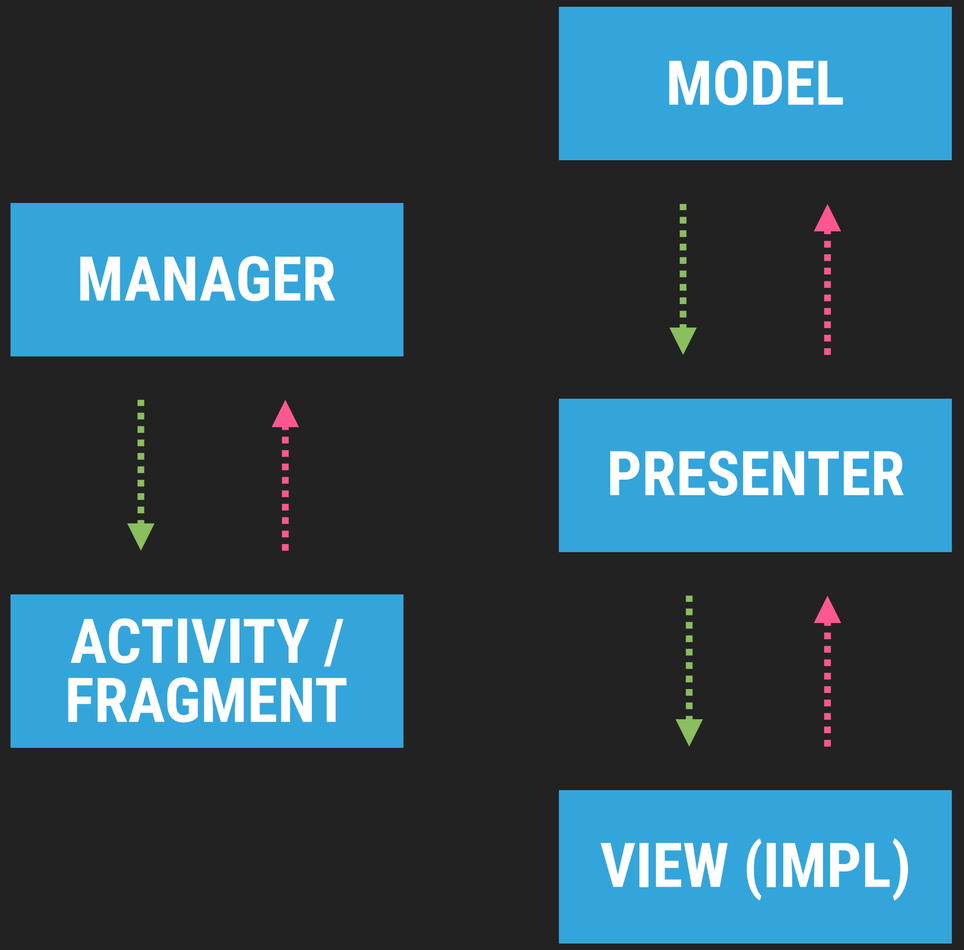
- Uncle Bob Martin

UMA APLICAÇÃO DOS DIAS ANTIGOS

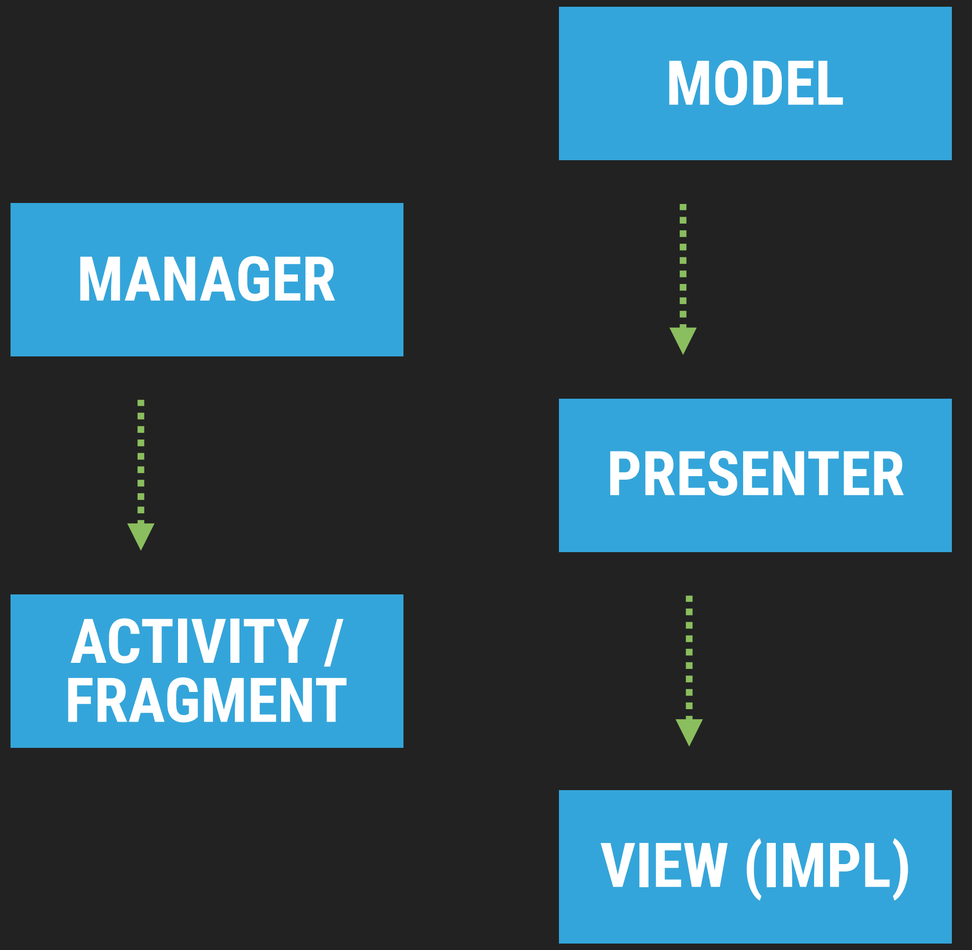


UMA APLICAÇÃO DOS DIAS MODERNOS

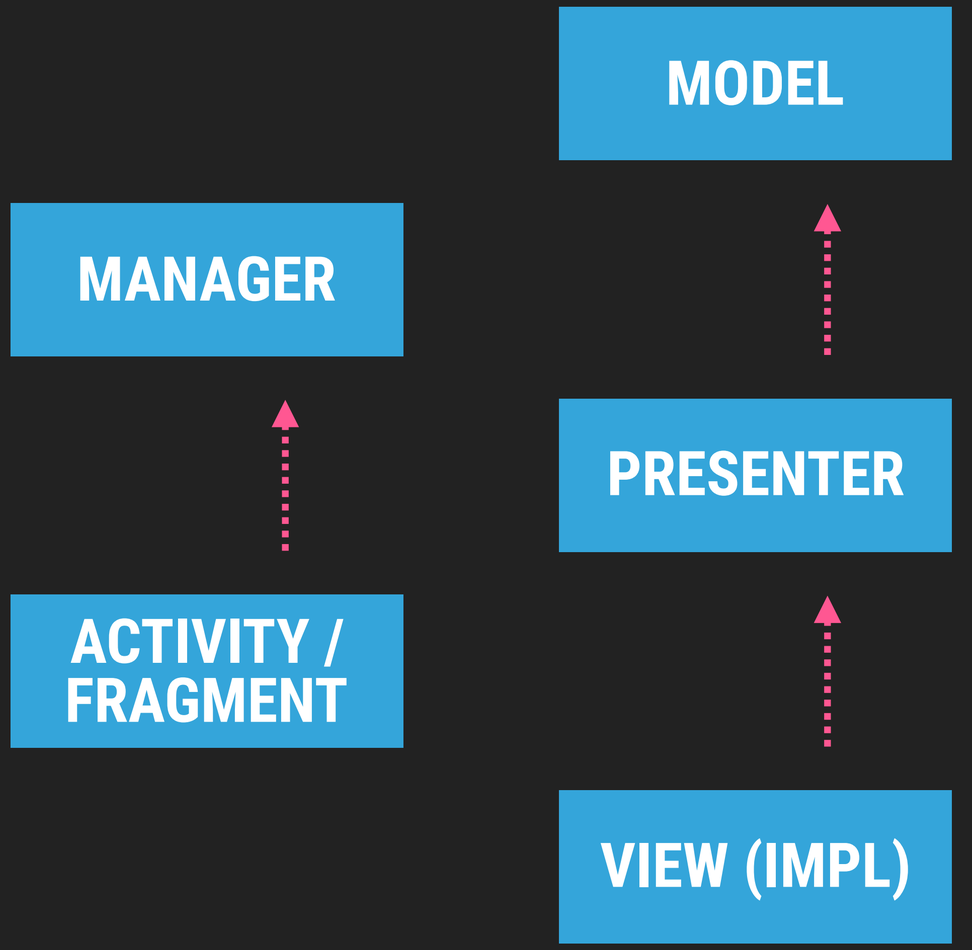




As direções de fluxo de dados indicam como você pode substituir callbacks / eventos por uma sequência observável!

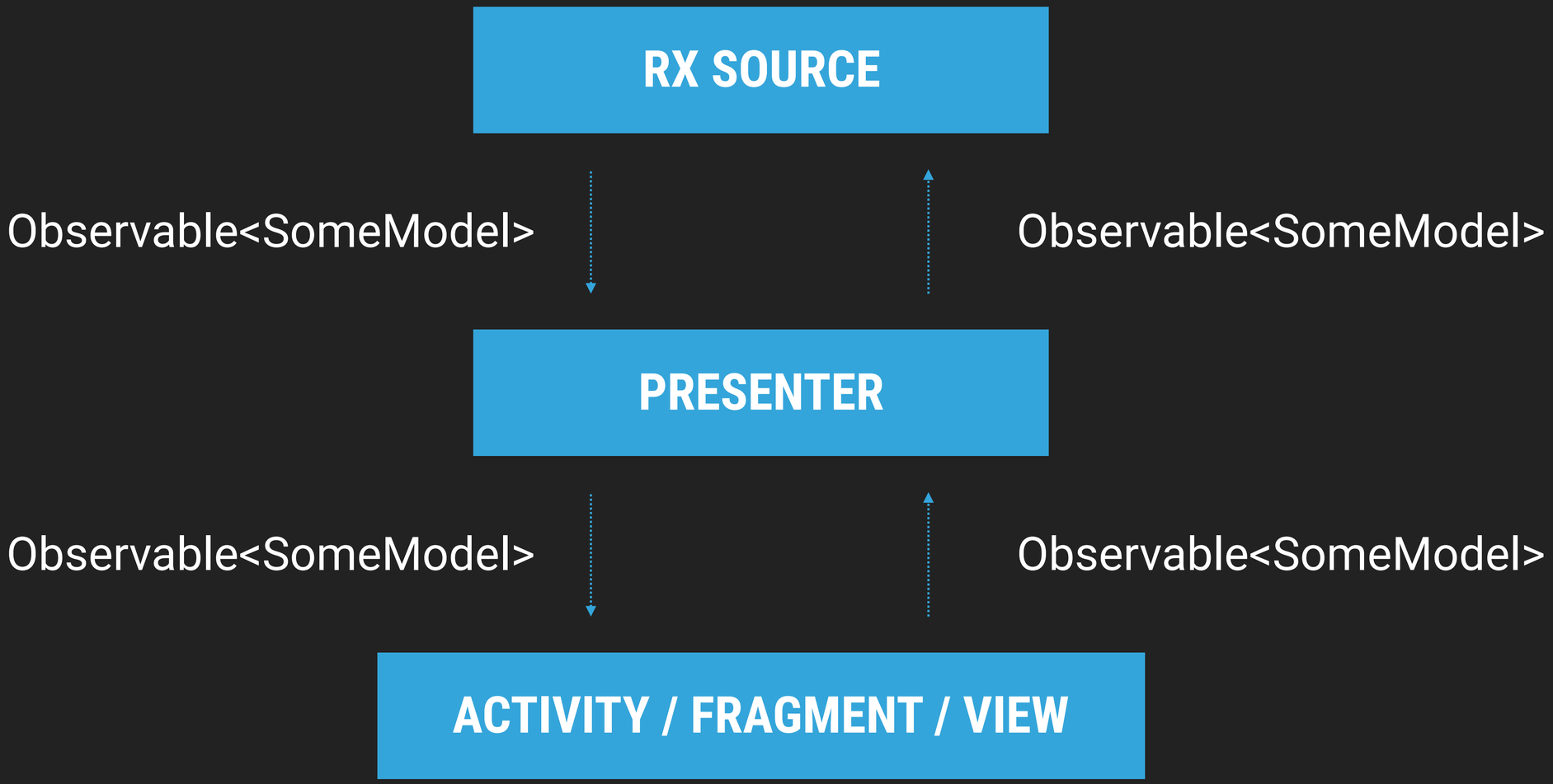


Caso 01 : as emissões serão geradas nas camadas mais internas da aplicação e consumidas nas camadas mais próximas à UI (Presenter ou Android)

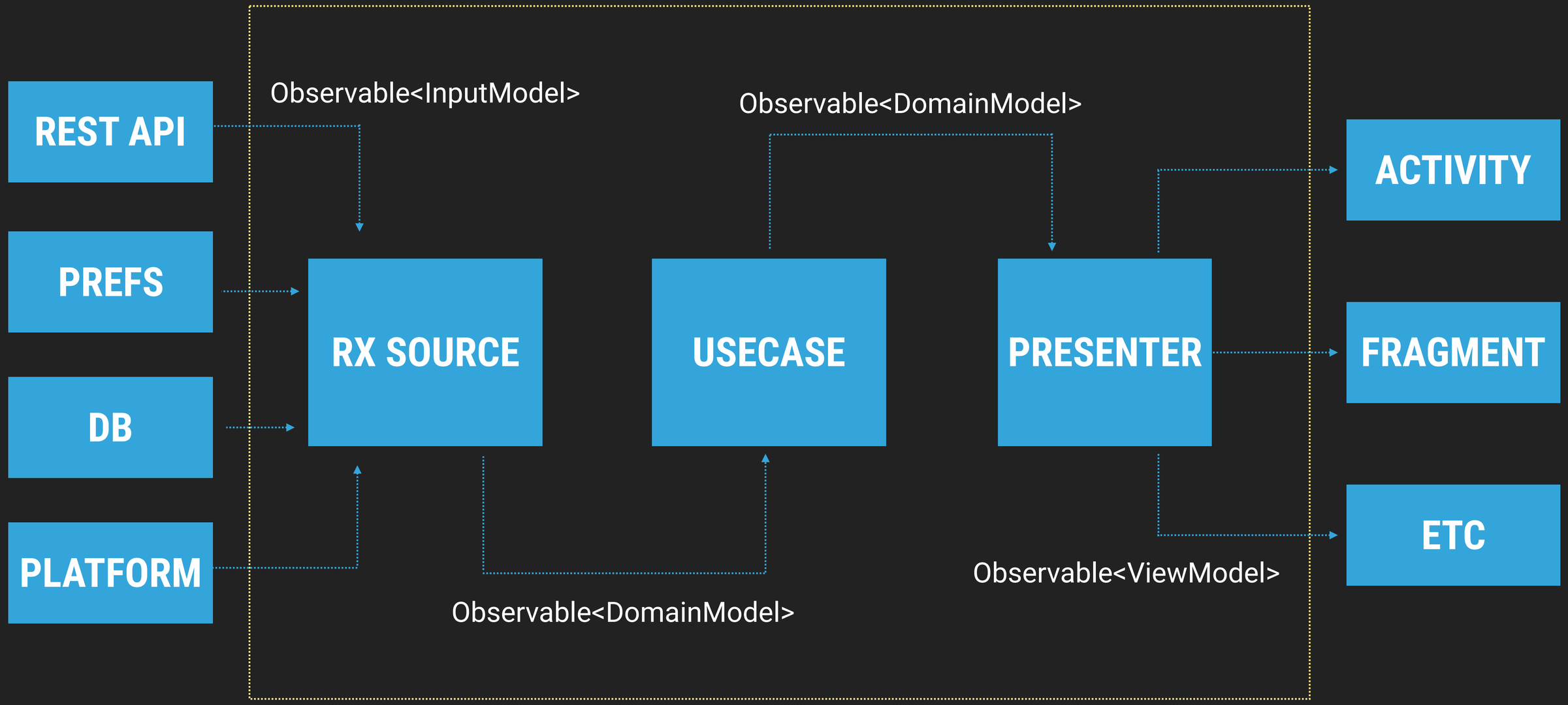


Caso 02 : as emissões são originadas da UI e consumidas nas camadas internas mais da aplicação

REACTIVE MVP



REACTIVE CLEAN ARCHITECTURE



CONSIDERAÇÕES PRÁTICAS

Consumir emissões no Presenter vs View Passiva ?

Consumir emissões no Android vs View Reativa ?

Consumo de emissões na UI vs Repository passivo(s) ?

Emissão na UI e consumo no Repository reativo?

Como lidar com estado no Presenter?

Como driblar *boilerplating* da replicação de dados?

Como testar tudo isso?

NON-OBVIOUS REACTIVE SOURCES

“Let`s catch them all”

Ash

ONDE PROCURAR?

Qualquer *callback* de uso recorrente pode ser encapsulado para emitir eventos em uma sequência observável

Android Framework está cheio deles!

APIs de suporte estão cheias deles!

PlayServices e adendos estão cheios deles!

ETC

```
class GoogleApiClientObservable extends BaseClient implements Action1<AsyncEmitter<GoogleApiClient>> {

    private final Api api;
    private AsyncEmitter<GoogleApiClient> emitter;

    private GoogleApiClientObservable(Context context, Api api) {
        super(context);
        this.api = api;
    }

    static Observable<GoogleApiClient> create(Context context, Api api) {
        return Observable.fromAsync(new GoogleApiClientObservable(context, api), BackpressureMode.NONE);
    }

    @Override public void call(AsyncEmitter<GoogleApiClient> emitter) {
        this.emitter = emitter;
        buildClient(api);
        connect();
        emitter.setSubscription(Subscriptions.create(this::disconnect));
    }

    @Override void onClientConnected(GoogleApiClient googleApiClient) {
        emitter.onNext(googleApiClient);
    }

    @Override void onClientError(Throwable throwable) {
        emitter.onError(throwable);
    }
}
```

Snippet from Servant

https://github.com/Mauin/servant

QUANDO NÃO USAR RXJAVA ?

“U HAVE NOTHING, NOTHING !!!!”

- Al Capone, The Untouchables

ALGUM CASOS A CONSIDERAR

Valores que não mudam nunca : justificam ser passados por Callback observável?

Observer/Subscriber desencadeia uma operação pesada no emissor, e a sequência por sua vez é multicasted

Você precisa de snapshots de estados intermediários referentes às emissões por algum motivo

Seu design de classes sugere que um Observable<T> até podia ser uma variável membro ...

ETC

FINAL REMARKS

"You know nothing, Jon Snow"

- Game of Thrones

DONT FORGET KIDS

Comece pelos casos simples

Evolua para os casos complexos

Defina quais camadas da sua aplicação são reativas ou não

Substitua callbacks/eventos por sequência observáveis

FTW

REFERÊNCIAS (I)

"Functional Reactive Programming with RxJava" by Ben Christensen
https://youtu.be/_t06LRX0DV0

“Learning RxJava (for Android) by example“ by Kaushik Goupal
https://youtu.be/k3D0cWyNno4

“Demystifying RxJava Subscribers" by Jake Wharton
https://youtu.be/NVKmyK6sd-Q

“What does it mean to be Reactive ?” by Erik Meijer
https://youtu.be/sTSQlYX5DU0

REFERÊNCIAS (II)

"Grokking RxJava Series" by Dan Lew
http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/

"The Introduction to Reactive Programming you`ve been missing" by André Staltz
https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

Oficial RxJava Wiki by NetFlix
https://github.com/ReactiveX/RxJava/wiki

Advanced RxJava Blog by David Karnok
akarnokd.blogspot.com

REFERÊNCIAS (III)

GradleLambda : https://github.com/evant/gradle-retrolambda

RxAndroid : https://github.com/ReactiveX/RxAndroid

RxLifecycle : https://github.com/trello/RxLifecycle

RxBinding : https://github.com/JakeWharton/RxBinding

Frodo : https://github.com/android10/frodo

speakerdeck.com/ubiratansoares/refactoring-to-rxjava

UBIRATAN SOARES

Computer Scientist by ICMC/USP

Software Engineer @ Luiza Labs

Google Developer Expert for Android

Teacher, speaker, etc, etc

THAT`S ALL FOLKS !!!

THANKS!

@ubiratanfsoares

br.linkedin.com/in/ubiratanfsoares

ubiratansoares.github.io