Moderne App-Entwicklung am Beispiel waipu.tv

Andreas Bauer

10.11.2016

Johannes Schamburger

Informationen zur inovex GmbH

- IT-Dienstleister in Pforzheim, Karlsruhe, Köln, München und Hamburg
- Application Development, Datamanagement & Analytics, Consulting und IT Engineering & Operations
- ca. 250 Mitarbeiter

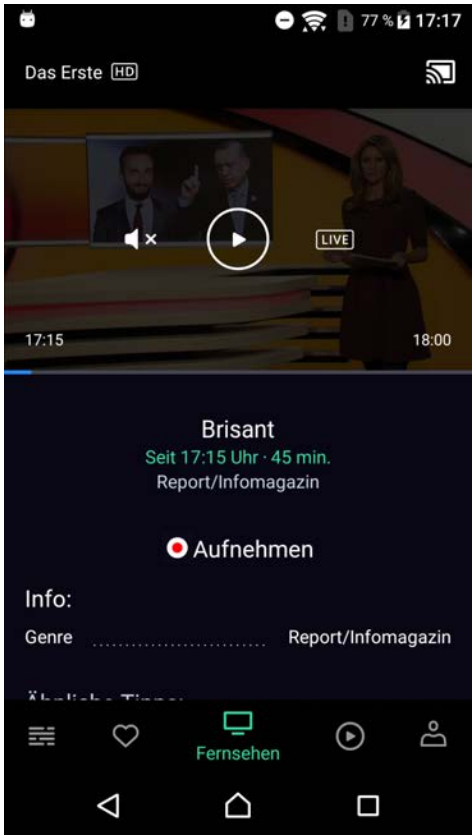
Informationen zu waipu.tv

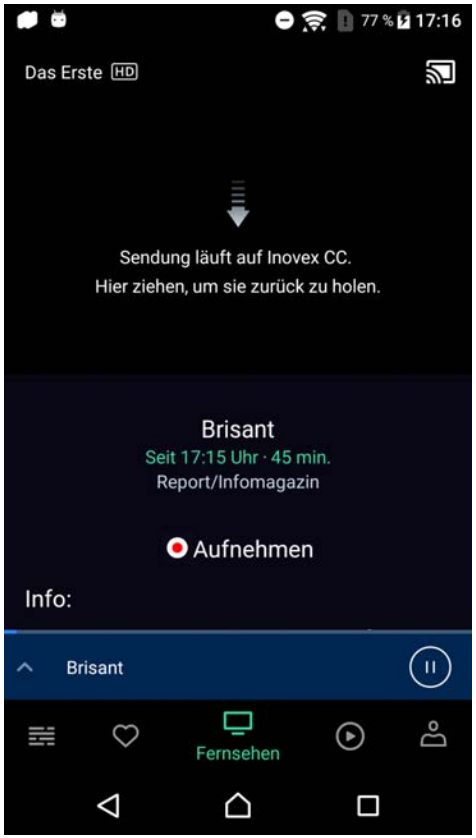
- Projekt der Firma EXARING AG
- Freenet ist Investor - Vermarktung u.a.
  in Mobilcom Debitel Läden
- Entwicklung gestartet im letzten Herbst
- Android Entwicklung seit Anfang des Jahres mit einer Teamgröße von 3-5 Entwicklern
- Launch von waipu.tv zum 1.10.2016

Was ist waipu.tv?

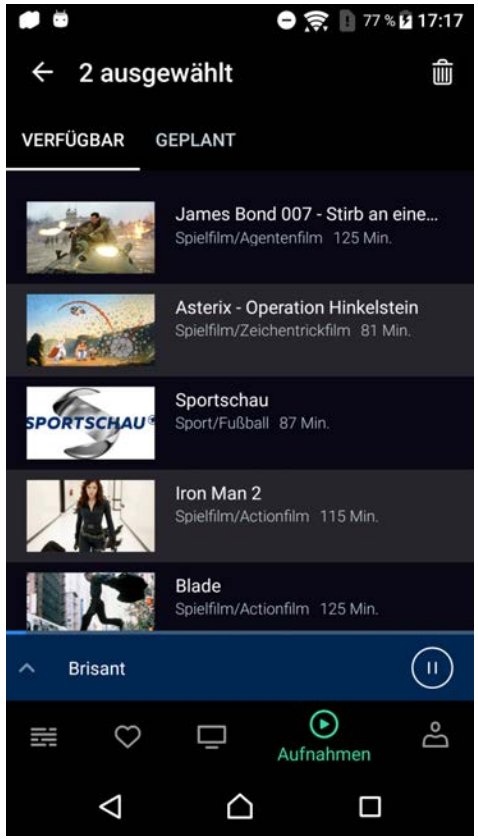
- TV Live-Streams
- Aufnehmen von TV-Streams
- Zeitversetzte TV-Streams
- Wiedergabe in der App und auf dem TV (Chromecast, Amazon Fire TV)
- App als (Next Generation) Fernbedienung

Streaming


77 % 17:17
Das Erste HD
LIVE
17:15
18:00
Brisant
Seit 17:15 Uhr · 45 min.
Report/Infomagazin
Aufnehmen
Info:
Genre
Report/Infomagazin
Fernsehen


77 % 17:16
Das Erste HD
Sendung läuft auf Inovex CC.
Hier ziehen, um sie zurück zu holen.
Brisant
Seit 17:15 Uhr · 45 min.
Report/Infomagazin
Aufnehmen
Info:
Brisant
Fernsehen

Weitere Funktionen



Motivation

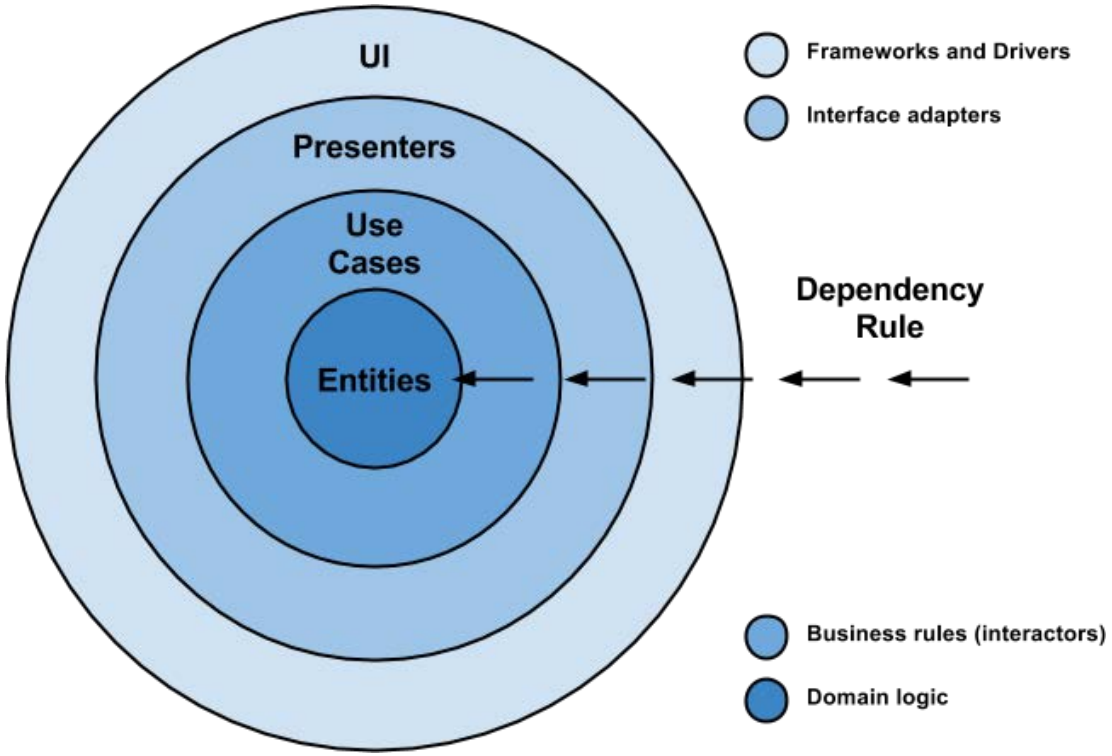
Typische Probleme in der Android App Architektur

- Gott-Klassen
- Testbarkeit
- Wartbarkeit
- Abhängigkeit vom Framework / Lifecycle

Überlegungen

- Welches Architekturmodell?
- Welche Libraries?
- Dependency Injection?
- Reactive Programming?

Clean Architecture (Uncle Bob)



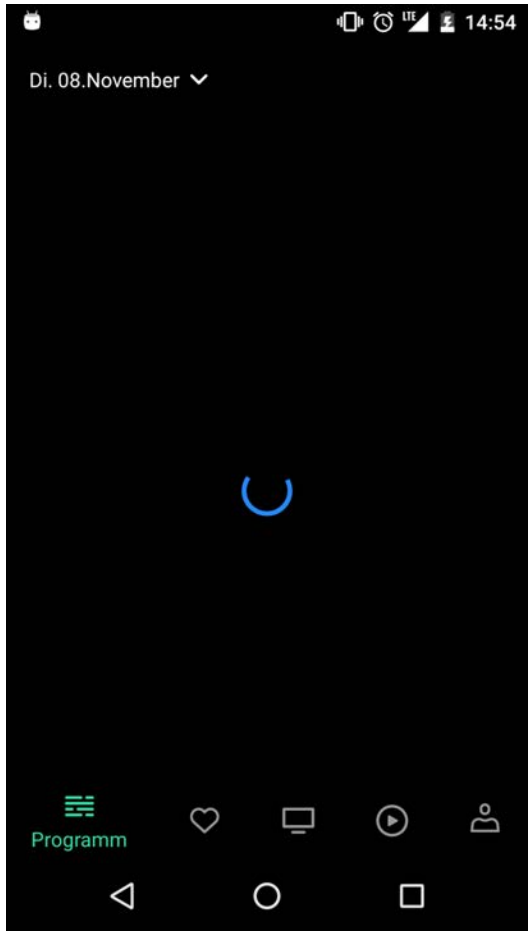
http://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/

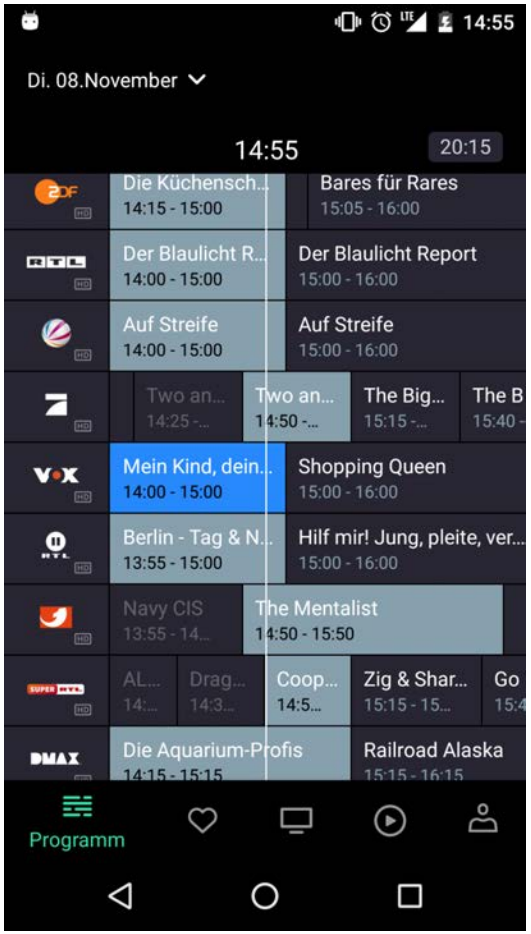
Eigenschaften

Die Architektur ist

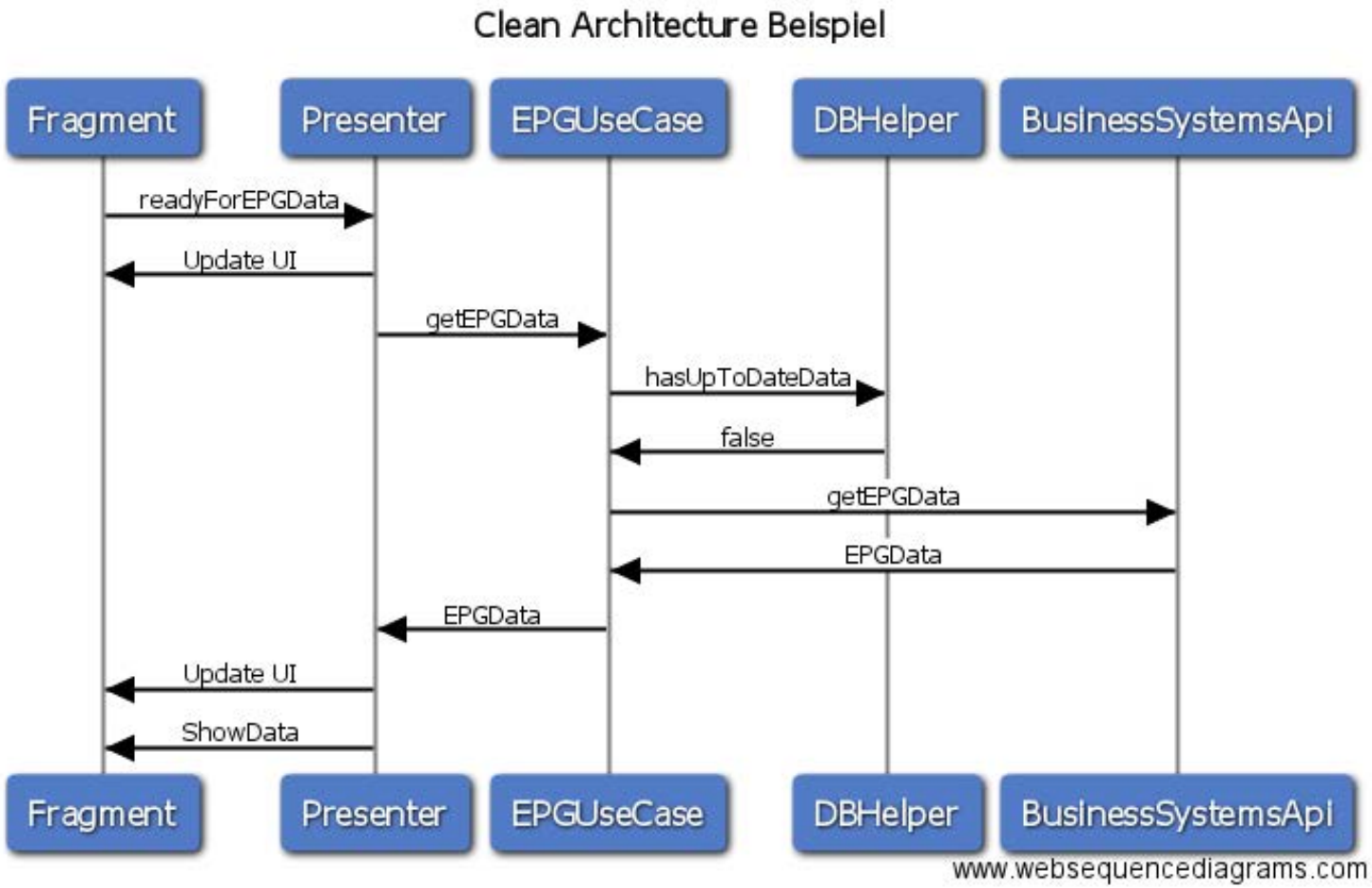
- unabhängig von der UI
- unabhängig von der Datenbank
- unabhängig von Frameworks
- testbar

Architektur Beispiel EPG Daten

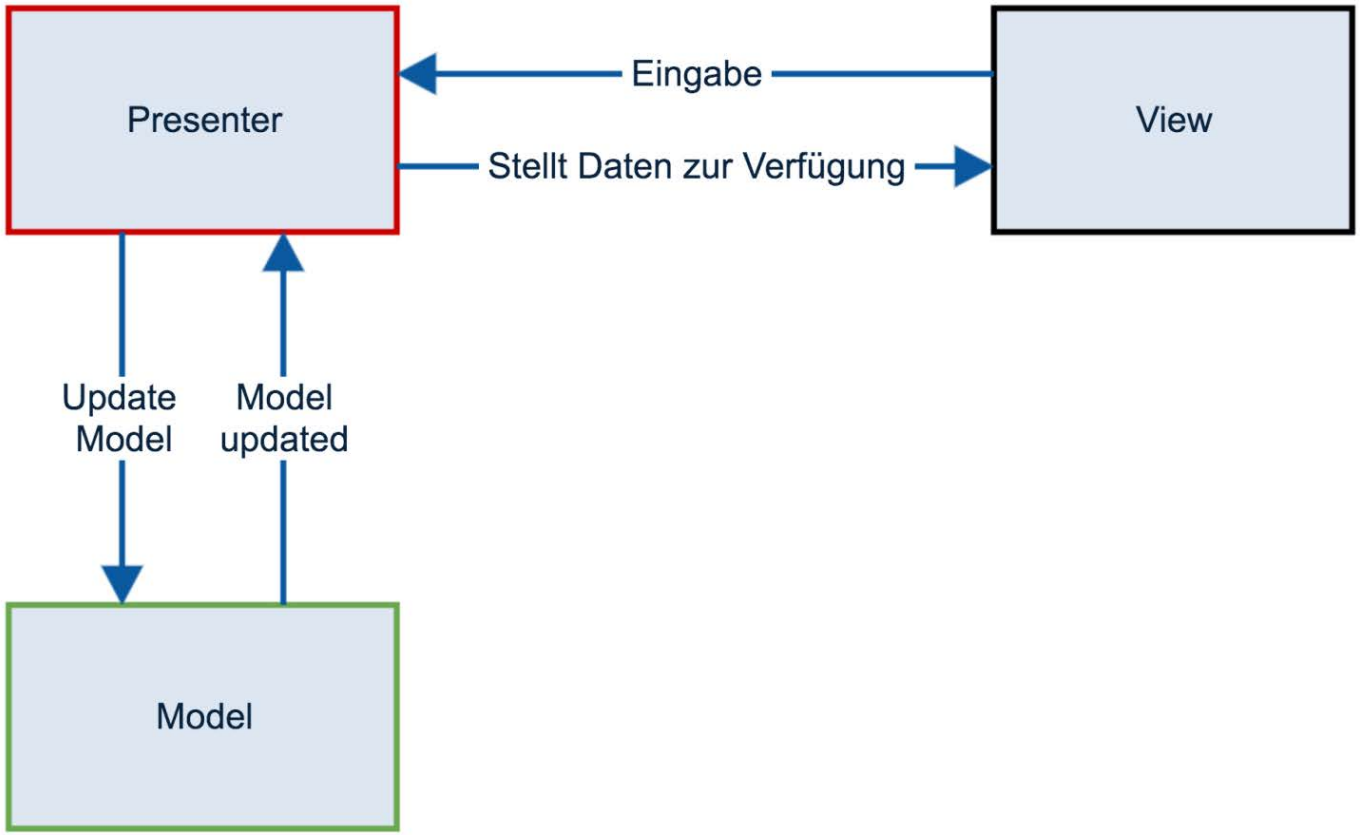




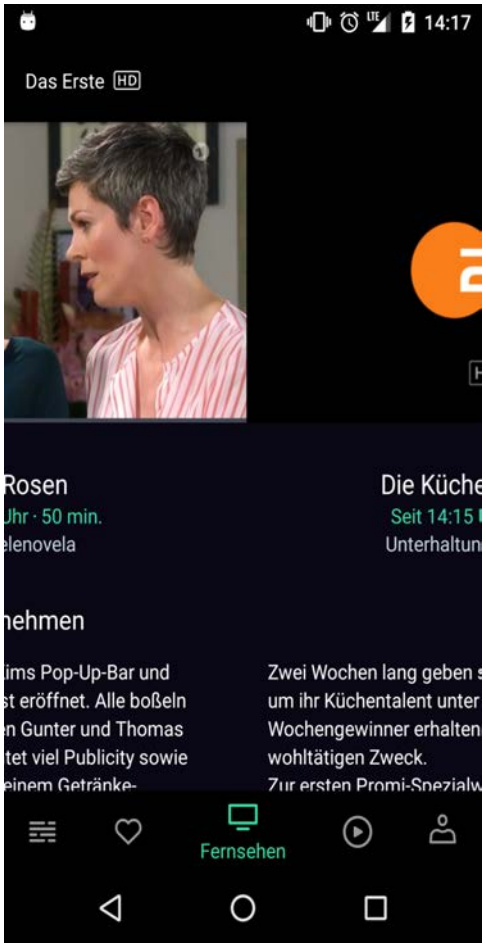
Architektur Beispiel EPG Daten



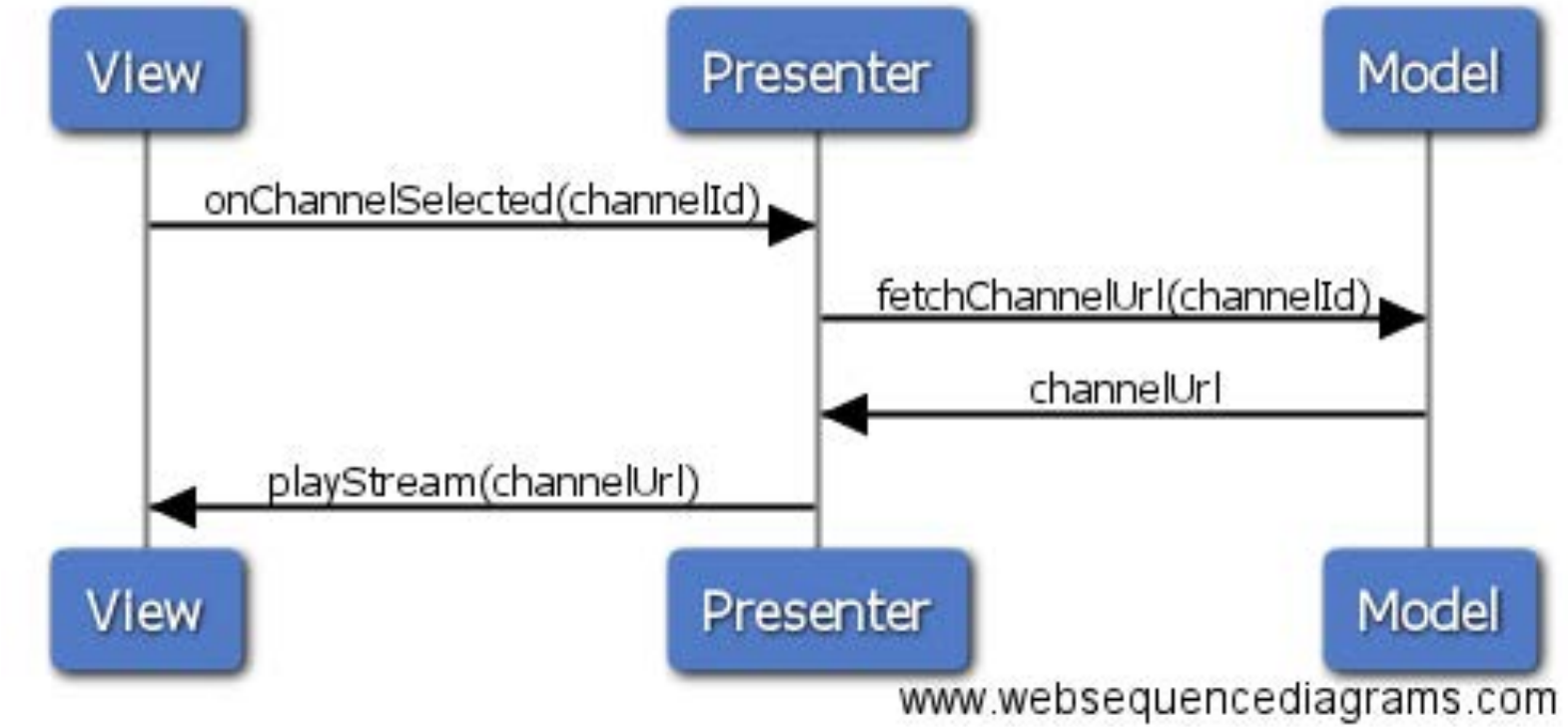
MVP - Model View Presenter



MVP Beispiel Kanalwechsel



MVP Beispiel Kanalwechsel



Presenter Beispiel

```
public interface StreamingPresenter<T> extends BasePresenter<T> {
   void onStop();
   void onDestroy();
   void onCastDeviceDisconnected();
   void onCastStarted();
   void onCastStopped();
   void onMuteToggled(boolean muted);
   void onJumpToStart();
   …
}
```

View Beispiel

```
public interface StreamingView<T> extends BaseView<T> {
   void setPreviewImage(String previewImageHref);
   void showErrorMessageForStream();
   void hideErrorMessageForStream();
   void setChannel(Channel channel);
   void updateCastOverlayMessage(String castDeviceName);
   void showFullScreenLoadingIndicator();
   void hideFullScreenLoadingIndicator();
   …
}
```

Pros und Cons von MVP

+ Separation of Concerns
+ Kleinere, übersichtlichere Klassen
+ Testbarkeit

- Klassen Overhead
- Mehraufwand in der Entwicklung
- saubere Umsetzung erzeugt teilweise lange Callstrecken

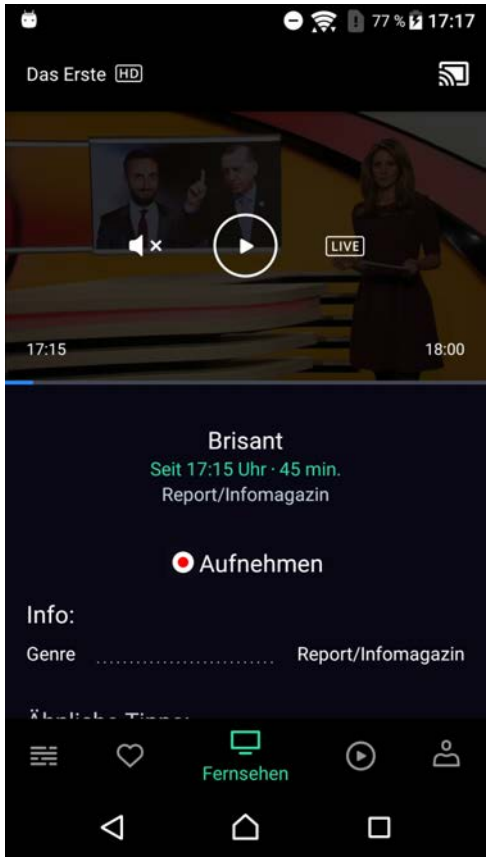
Beispiel Presenter Unit-Test

```
@Test
public void select_channel_should_start_stream() throws Exception {
   String streamUrl = "streamUrl";
   doReturn(Observable.just(streamUrl)).when(streamUseCase)
                                .getStreamURL(anyString(), anyString(), anyString(), anyString(), anyString(), anyLong());
   presenter.onLocalChannelSelected("ARD");
   verify(liveTvView).hideErrorMessageForStream();
   verify(liveTvView).setupVideoPlayer(any(VideoPlayer.class), any(EventLogger.class));
   verify(liveTvView).setStream(streamUrl, VideoPlayerView.StreamType.TYPE_DASH);
   verify(liveTvView).startDevicePlayback();
   verify(liveTvView).updateCastViewState();
   verifyNoMoreInteractions(liveTvView);
}
```

Beispiel lange Callstrecke

Szenario: Sender wird gewechselt, Sendungsdetails werden aktualisiert.

Container Fragment -> LiveTvFragment -> LiveTvPresenter -> LiveTvFragment -> TvDetailsView -> TvDetailsPresenter -> TvDetailsView



Reactive Programming

“In reactive programming the consumer reacts to the data as it comes in. This is the reason why asynchronous programming is also called reactive programming. Reactive programming allows to propagates event changes to registered observers.” - Lars Vogel

RxJava

- Java VM Implementierung der ReactiveExtensions
- Portiert von .NET zu JVM von Netflix 2014
- leichtgewichtig
- Open Source
- Seit 2014 im ReactiveX Repository auf GitHub

Einfaches Beispiel Observable

```
Observable<String> simpleObservable = Observable.create(
        new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<? super String> subscriber) {
                subscriber.onNext("Simple Value");
                subscriber.onCompleted();
            }
        });
```

Einfaches Beispiel Subscriber

```
Subscriber<String> simpleSubscriber = new Subscriber<String>() {
   @Override
   public void onNext(String value) {
      println(value);
   }

   @Override
   public void onCompleted() {}

   @Override
   public void onError(Throwable e) {}
};
simpleObservable.subscribe(simpleSubscriber);
```

Operatoren manipulieren emittierte Items

```
Observable<String> simpleObservable = Observable.just("Simple Value")
        .map(new Func1<String, String>() {
            @Override
            public String call(String value) {
                return value + " now modified";
            }
        });

simpleObservable.subscribe (new Subscriber<String>() {
   @Override
   public void onNext(String value) {
       println(value);  // prints “Simple Value now modified"
   }
…  }
```

Rx bei waipu.tv

```
public Observable<List<Channel>> getChannels() {
  Observable<List<Channel>> apiCall = authorization.
        .getAuthorizationStringAsObservable()
        .flatMap(new Func1<String, Observable<? extends List<Channel>>>() {
           @Override
           public Observable<? extends List<Channel>> call(String auth) {
              return businessSystemsApi.getChannelData(auth);
           }
        });
```

- Authentifizierung

- BS - Aufruf

```
  return authorization.loginWhenRequired(apiCall)
        .doOnNext(new Action1<List<Channel>>() {
           @Override
           public void call(List<Channel> channels) {
              dbHelper.insertChannelList(channels);
           }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());    }
```

- Zusätzlicher Login
- Retry
- Datenbankoperation

- Thread Handling

Rx bei waipu.tv

```
remoteDataSubject.observeOn(AndroidSchedulers.mainThread())
      .map(new Func1<Void, Boolean>() {
        @Override
                public Boolean call(Void aVoid) {
          // get Current Remote Program Information
          // if program is still valid return false
          // else return true
        } })
```

```
 .distinctUntilChanged()
      .observeOn(Schedulers.io())
      .flatMap(new Func1<Boolean, Observable<List<EPGData>>>() {
        @Override
        public Observable<List<EPGData>> call(Boolean value) {
          // Return Observable which emits Program Data for a timespan
        }
      })
```

Rx bei waipu.tv

```
.flatMap(new Func1<List<EPGData>, Observable<ProgramInformation>>() {
    @Override public Observable<ProgramInformation> call(List<EPGData> epg) {
        // Get Current Timestamp
        // return ProgramOverview for channel
    }
})


.retry(new Func2<Integer, Throwable, Boolean>() {
    @Override public Boolean call(Integer retryCount, Throwable t) {
        return retryCount < 3;
    }
})
.subscribe(new DefaultSubscriber<ProgramInformation>("NotificationModelUpdater") {
    @Override public void onNext(ProgramInformation programInformation) {
        // Update Program Information
    }
});
```

Problematiken

```
private Subscription statusSubscription;
private void listenToStatusChanges() {
  statusSubscription = someObject.listenToStatusChanges()
        .subscribe(new Subscriber<StatusObject>() {
            ...
            @Override
            public void onNext(StatusObject o) {
               //do Stuff
            }  });
}

private void unsubscribeToStatusChanges() {
  if(statusSubscription != null && !statusSubscription.isUnsubscribed()) {
     statusSubscription.unsubscribe();
  }
}
```

Dependency Injection

- Klassen verwalten ihre Abhängigkeiten nicht selbst
- Single Responsibility
- komfortable Möglichkeit, Singletons zu verwenden
- einfaches Austauschen von konkreten Implementierungen (z.B. für Tests)

Dependency Injection - Dagger

Inject

```
@Inject
StreamUseCase streamUseCase;
```

Dependency Injection - Dagger

Module, Provides, Singleton

```
@Module
public class UseCaseModule {


   @Provides
   @Singleton
   public StreamUseCase provideStreamUseCase(BusinessSystemsApi
businessSystemsApi, AuthUseCase authUseCase, EPGUseCase epgUseCase) {
       return new StreamUseCase(businessSystemsApi, authUseCase, epgUseCase);
   }
}
```

Dependency Injection - Dagger

Component

```
@Component(modules = {AppModule.class, CastModule.class, UseCaseModule.class})
public interface AppComponent {
   void inject(WaipuApplication application);


   StreamUseCase streamUseCase();
}
```

Dependency Injection - Dagger

Component

```
@Component(modules = {AppModule.class, CastModule.class, UseCaseModule.class})
public interface AppComponent {
   void inject(WaipuApplication application);


   StreamUseCase streamUseCase();
@Component(dependencies = AppComponent.class, modules = {LiveTvModule.class})
public interface LiveTvComponent {
   void inject(LiveTvFragment fragment);
   void inject(LiveTvPresenterImpl liveTvPresenter);
}
```

Dependency Injection - Dagger

Injection

```
public class LiveTvFragment implements LiveTvView {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    appComponent = WaipuApplication.get(getActivity()).getAppComponent();
    liveTvComponent = DaggerLiveTvComponent.builder()
                          .appComponent(appComponent)
                          .liveTvModule(new LiveTvModule(this))
                          .build();
    liveTvComponent.inject(this);
  }
}
```

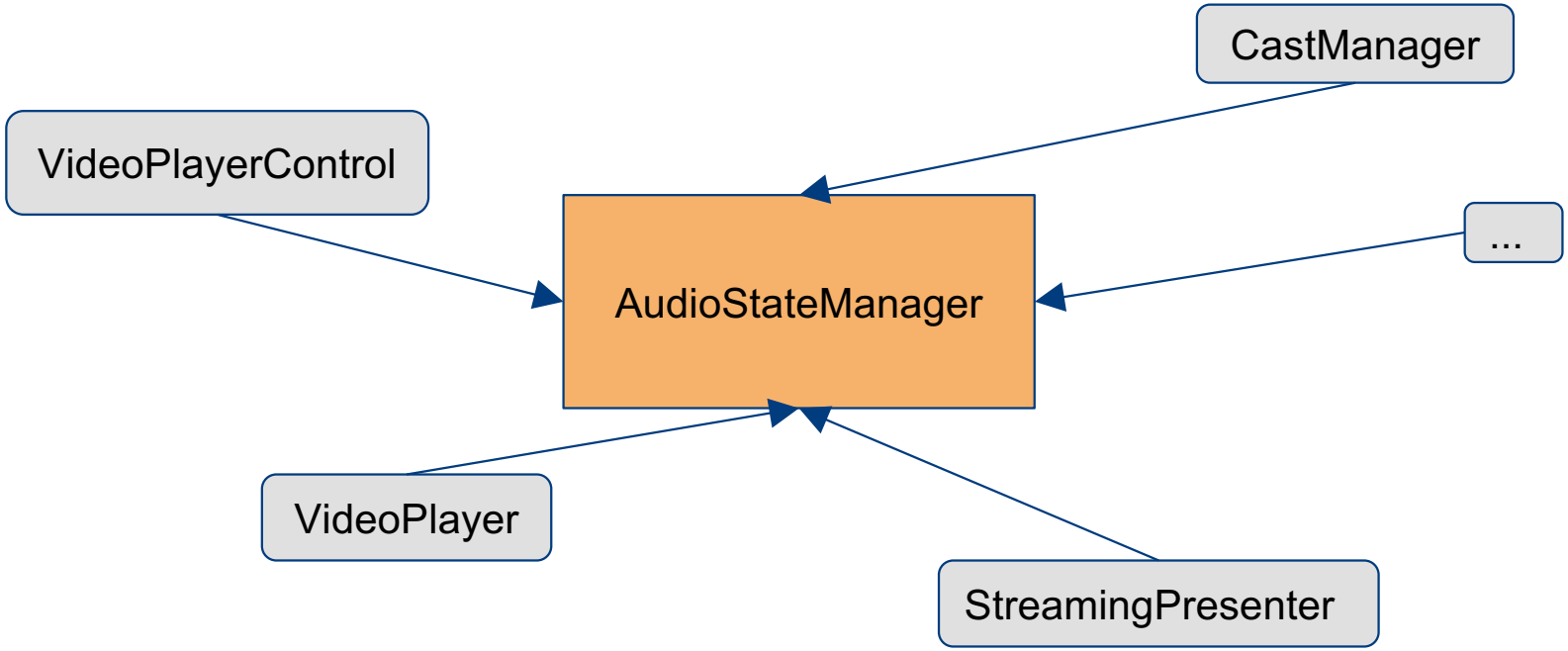
Pros and Cons

+ Instanziierung und Konfiguration gut nachvollziehbar
+ Singletons
+ Testing

- Klassen-Overhead
- “Boilerplate” Code

Singletons

```
@Inject
protected AudioStateManager audioStateManager;
```



Klassen-Overhead & Boilerplate Code

Klassen-Overhead & Boilerplate Code

LiveTvComponent:

```
@FragmentScope
@Component(dependencies = AppComponent.class, modules = {LiveTvModule.class})
public interface LiveTvComponent {
   void inject(LiveTvFragment fragment);
   void inject(LiveTvPresenterImpl liveTvPresenter);
}
```

Klassen-Overhead & Boilerplate Code

LiveTvComponent:

```
@FragmentScope
@Component(dependencies = AppComponent.class, modules = {LiveTvModule.class})
public interface LiveTvComponent {
  void inject(LiveTvFragment fragment);
  void inject(LiveTvPresenterImpl liveTvPresenter);
}
```

LiveTvFragment:

```
@Override
public void setupComponent(AppComponent appComponent) {
  liveTvComponent = DaggerLiveTvComponent.builder()
                                .appComponent(appComponent)
                                .liveTvModule(new LiveTvModule(this))
                                .build();
  liveTvComponent.inject(this);
}
```

Fazit und Erkenntnisse

- Mehraufwand zahlt sich langfristig aus
- Herausforderung: konsequente Umsetzung
- automatisiertes Testen bei Android nach wie vor umständlich
- RxJava - easy to learn, hard to master
- Gefahr von sehr großen Klassen ist nicht gebannt

Vielen Dank

Unser Dank gilt besonders der EXARING AG, die uns diesen Vortrag ermöglicht hat.