inovex

Moderne Android-App - Aber wie?

Daniel Bälz

Karlsruhe, 21.11.2016

“Greetings, Exalted One. Allow me to introduce myself.”

Star Wars: Episode VI - Return of the Jedi

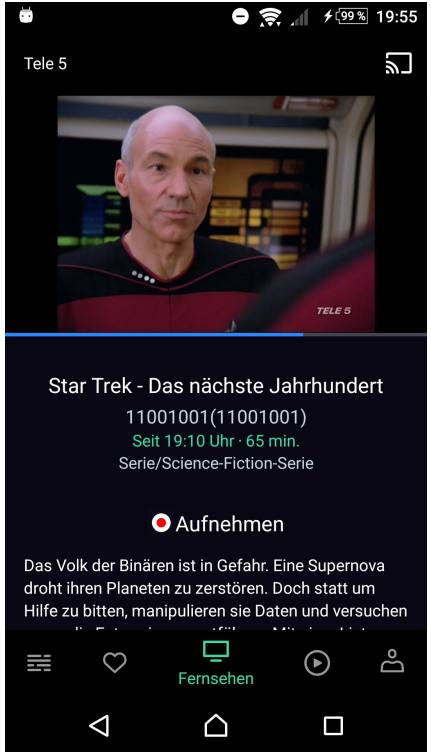
waipu.tv

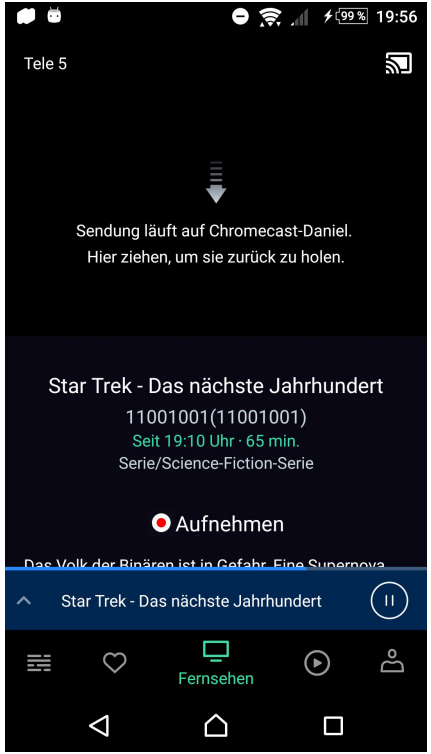
- *„Fernsehen wie noch nie”*
- Produkt der EXARING AG
- Android-App: Umsetzung seit Januar 2016 mit 2-5 Entwicklern
- Launch: Oktober 2016

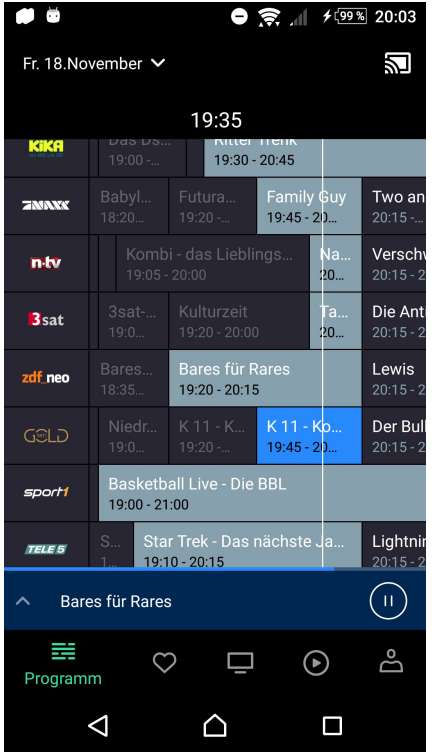
Feature

- TV Livestreams
- Zeitversetzte Streams
- Aufnehmen von Sendungen
- Wiedergabe per App und Chromecast (bald: Fire TV)
- App als erweiterte Fernbedienung
- Programmübersicht
- Persönliche Tipps

Demo Android-App







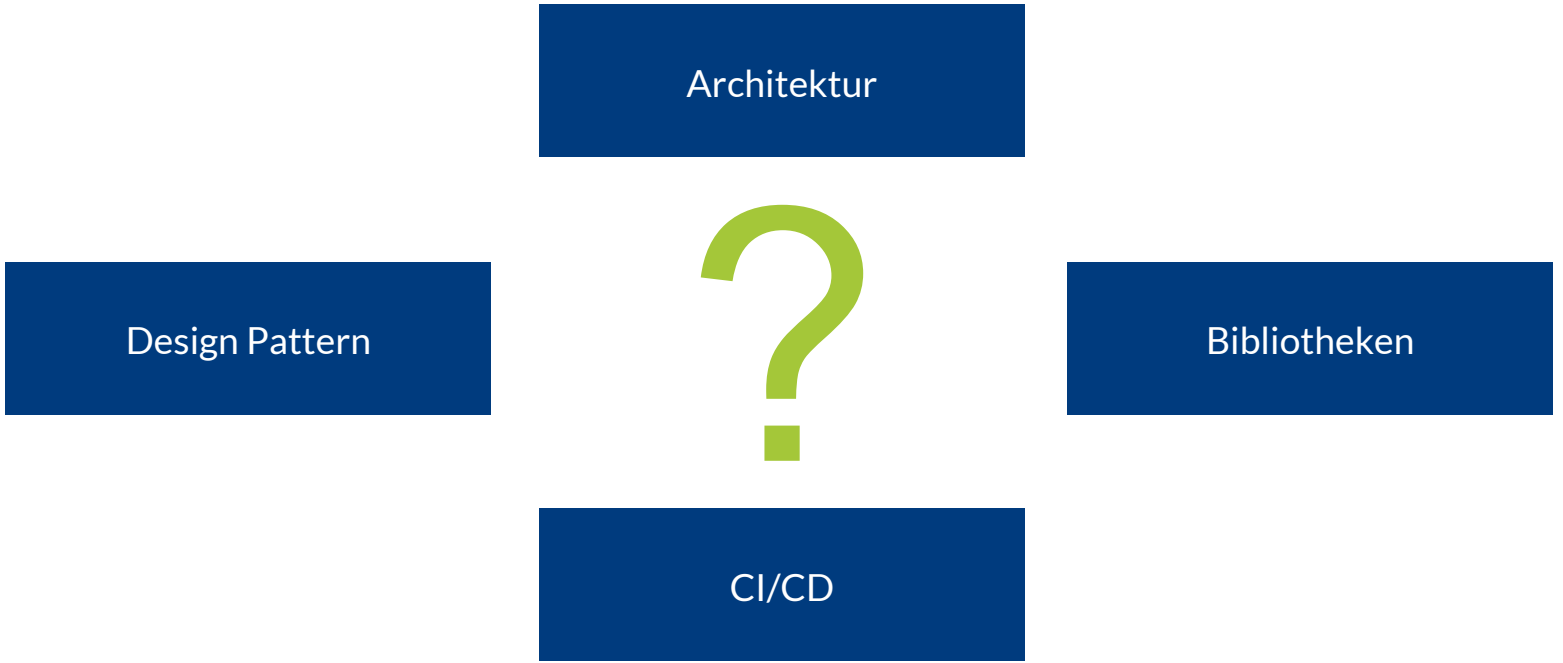
“Roads? Where we're going we don't need roads.”

Back to the Future

Motivation

- Gute Grundstruktur bereitstellen
- Weiterentwicklung vereinfachen
- Wachsende Anforderungen einplanen
- Testbarkeit ermöglichen
- Trends bei Android einfach integrieren

Überlegungen



Herausforderungen

- Android Lifecycle
- Activity/Fragment Gottklasse
- Abhängigkeiten vom API Level
- Kommunikation innerhalb der App
- Externe, asynchrone Kommunikation
- Testbarkeit (Instrumentation und lokal ausführbar)

“I see in your eyes the same fear that would take the heart of me”

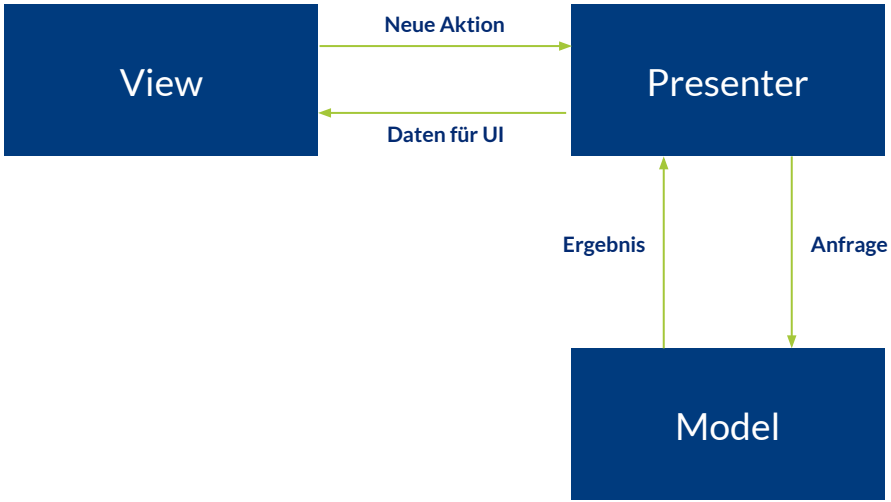
The Lord of the Rings: The Return of the King

Model-View-Presenter

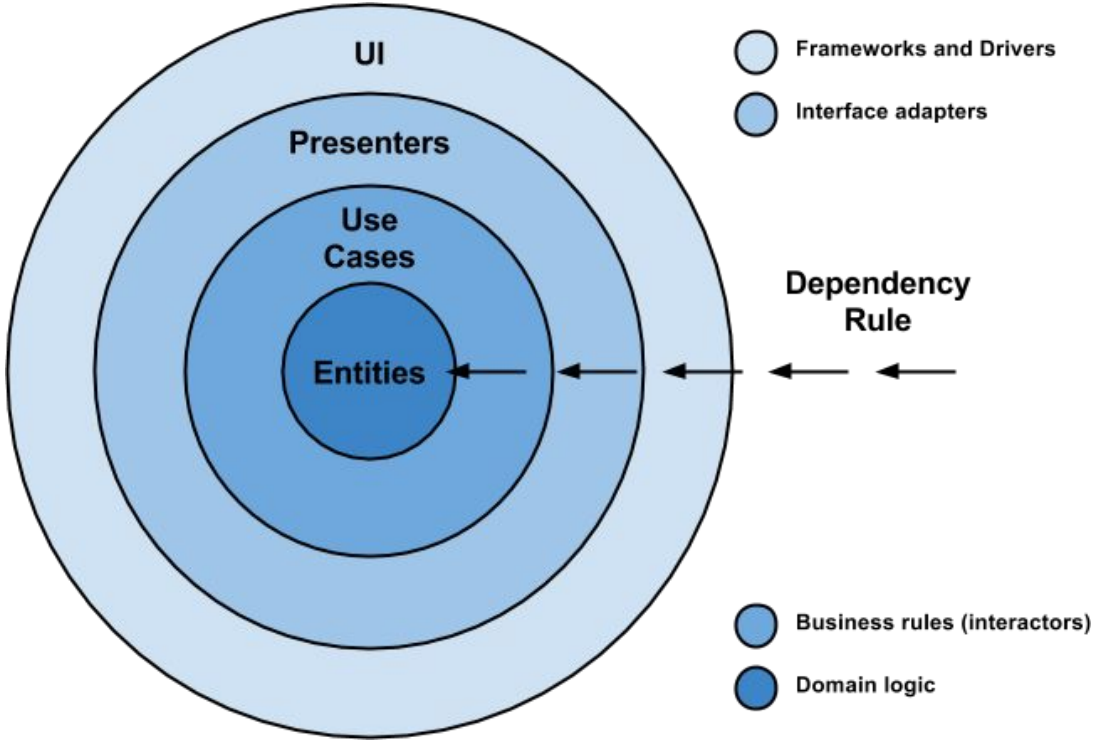
- Model: Geschäftslogik (Anbindung Backend)
- View: Darstellung (Fragment)
- Presenter: Steuerung des Ablaufs

- Umsetzung
  - Mit Bordmitteln
  - Definition über Interfaces

Model-View-Presenter

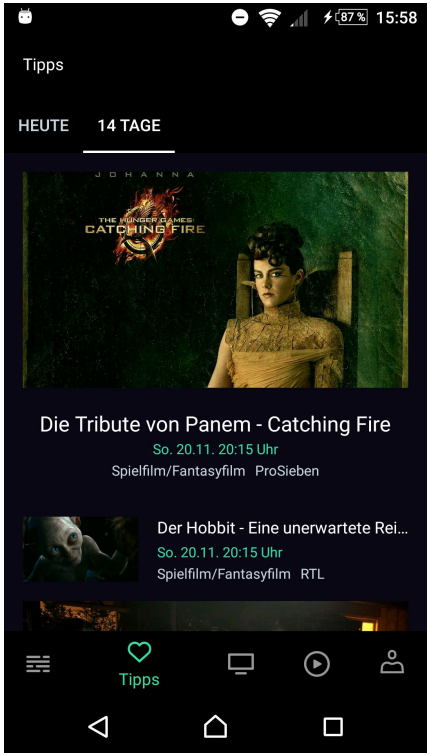


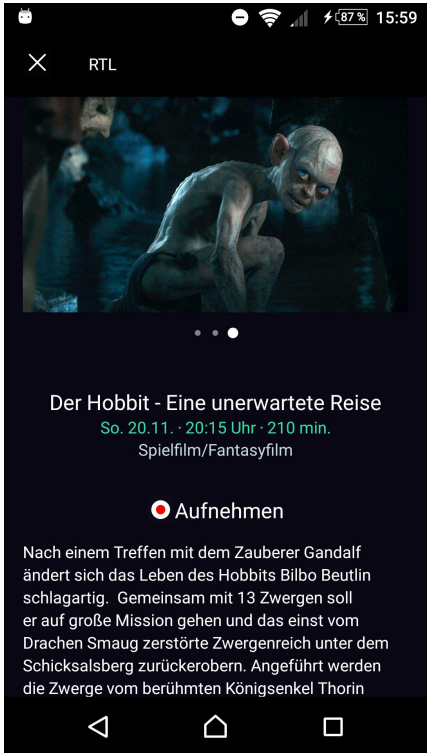
Clean Architecture



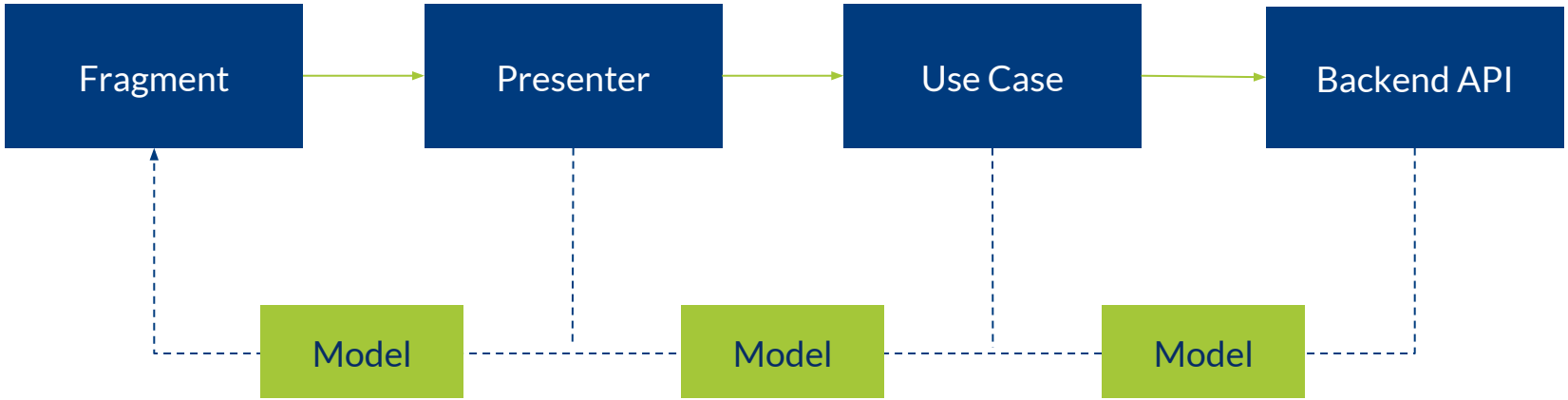
Quelle: http://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/

Beispiel Verwendung





Beispiel Verwendung



Vor- und Nachteile

- Vorteile
  - Gute Trennung (Separation of Concerns)
  - Übersichtlichere Klassen
  - Testbarkeit der einzelnen Komponenten
- Nachteile
  - Mehraufwand bei der Umsetzung
  - Mehr Klassen
  - Lange Aufrufpfade

Reactive Extensions

„ReactiveX is a library for composing asynchronous and event-based programs by using observable sequences.”

Quelle: http://reactivex.io/intro.html

RxJava

- Implementiert der Reactive Extensions für Java
- Open Source
- Android
  - RxAndroid: Bindings für Android
  - RxBinding: UI Widget Bindings

Bestandteile

- Observable
  - Emittiert Daten
  - Thread Handling
- Subscriber
  - Konsumiert Daten
- Operator
  - Manipuliert Daten
- Subject
  - Observer und Observable

Umfang

- Viele Arten Observable zu erzeugen
- Verschiedene Subscriber
- Sehr (sehr) viele Operatoren
- Mehrere Subjects

Subscriber

```
Subscriber<Integer> subscriber = new Subscriber<Integer>() {
  @Override
  public void onCompleted() {
  }

  @Override
  public void onError(Throwable e) {
  }

  @Override
  public void onNext(Integer integer) {
  }
};
numberObservable.subscribe(subscriber);
```

Observable

```
Observable<Integer> numberObservable =
    Observable.create(new Observable.OnSubscribe<Integer>() {
      @Override
      public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(42);
        subscriber.onCompleted();
      }
    });
```

Thread Handling

```
Observable<Integer> numberObservable =
    Observable.create(new Observable.OnSubscribe<Integer>() {
      @Override
      public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(42);
        subscriber.onCompleted();
      }
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread());
```

Operator

```
Observable<Integer> numberObservable =
    Observable.create(new Observable.OnSubscribe<Integer>() {
      @Override
      public void call(Subscriber<? super Integer> subscriber) {
        subscriber.onNext(42);
        subscriber.onCompleted();
      }
    }).map(new Func1<Integer, Integer>() {
      @Override
      public Integer call(Integer integer) {
        return integer * 2;
      }
    });
```

Subject

```
PublishSubject<Integer> publishSubject = PublishSubject.create();
publishSubject.subscribe(subscriber);
publishSubject.onNext(0);
publishSubject.onNext(1);
publishSubject.subscribe(subscriber2);
publishSubject.onNext(2);
publishSubject.onNext(3);
publishSubject.onCompleted();

BehaviorSubject<Integer> behaviorSubject = BehaviorSubject.create(-1);
behaviorSubject.onNext(0);
behaviorSubject.onNext(1);
behaviorSubject.subscribe(subscriber3);
behaviorSubject.onNext(2);
behaviorSubject.onNext(3);
behaviorSubject.onCompleted();
```

Beispiel aus waipu.tv

```
public Observable<List<Channel>> getChannels() {
  Observable<List<Channel>> apiCall = authUseCase
        .getAuthorizationStringAsObservable()
        .flatMap(new Func1<String, Observable<? extends List<Channel>>>() {
          @Override
          public Observable<? extends List<Channel>> call(String auth) {
            return businessSystemsApi.getChannelData(auth);
          }
        });
  return authUseCase.loginWhenRequired(apiCall)
        .doOnNext(new Action1<List<Channel>>() {
          @Override
          public void call(List<Channel> channels) {
            dbHelper.insertChannelList(channels);
          }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());
}
```

Vor- und Nachteile

- Vorteile
  - Thread Handling
  - Datenmanipulation
  - Komplexe Szenarien lösbar
- Nachteile
  - Verwalten der Subscription
  - Gefahr von Memoryleaks
  - Lernkurve

Dagger 2

- Bibliothek für Dependency Injection
  - Single Responsibility
  - Abhängigkeiten in zentraler Klasse verwaltet
  - Singleton
- Codegenerierung zur Compile-Zeit
- Komponenten und Modulsystem

Module

```
@Module
public class BusinessSystemsModule {

  @Provides
  @Singleton
  public RecommendationsUseCase provideRecommendationsUseCase(
                                                    BusinessSystemsApi businessSystemsApi,
                                                    AuthUseCase authUseCase) {
    return new RecommendationsUseCase(businessSystemsApi, authUseCase);
  }
}
```

Component

```
@Component(modules = {SharedPreferencesModule.class, BusinessSystemsModule.class})
public interface AppComponent {
  void inject(WaipuApplication application);
  void inject(BaseActivity baseActivity);

  RecommendationsUseCase recommendationsUseCase();
}
```

Injection

```
@Inject
RecommendationsUseCase recommendationsUseCase;
```

Vor- und Nachteile

- Vorteile
  - Klare Struktur der Abhängigkeiten
  - Einfache Verwendung
  - Austauschbarkeit für Tests
- Nachteile
  - Mehr eigene Klassen
  - Viele generierte Klassen

Weitere Bibliotheken

- Support Library
- Retrofit
- Glide
- Timber
- Butterknife
- Joda Time
- Stetho

“[...] and I show you how deep the rabbit hole goes.”

The Matrix

Entwicklung

- Scrum
- Wissenstransfer
  - Kein Inselwissen
  - Merge Request mit intensivem Review
  - Diskussionen
  - Dokumentation
- Kritische Betrachtung des aktuellen Stands
- Bereitschaft zur Veränderung

CI/CD

- Gitflow
- Gitlab CI
  - YAML zur Jobbeschreibung
  - Docker Images
  - Testausführung
  - Upload des Artefakt
- Reporting Crash und Fehler

Fazit

- Initialen Mehraufwand nicht scheuen
- Konsequente Umsetzung lohnt sich
- Vorhandene Bibliotheken nutzen
- Bibliotheken/Abhängigkeiten erzeugen neue Probleme
- Automatisiertes Testen immer noch umständlich

Vielen Dank