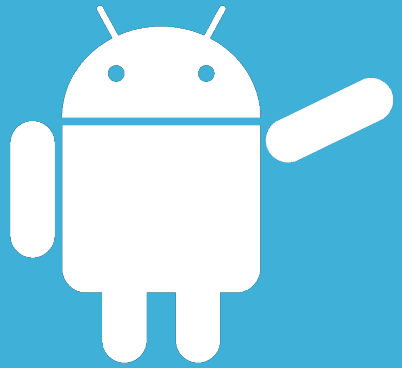
FORTGESCHRITTENE ANDROID ENTWICKLUNG

Max Wielsch

27.05.2015

Saxonia Systems

So geht Software.



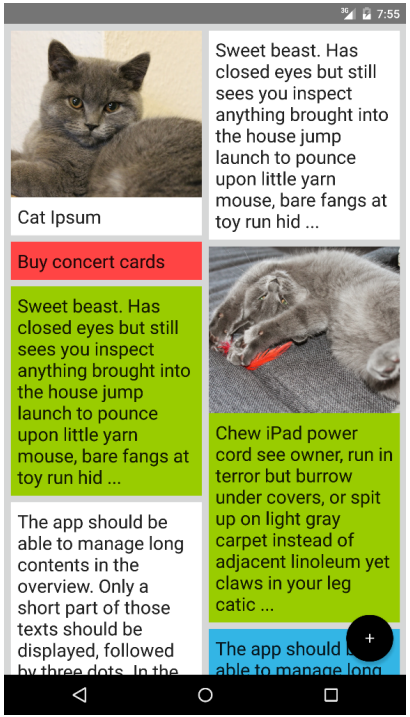
WAS LETZTES MAL GESCHAH

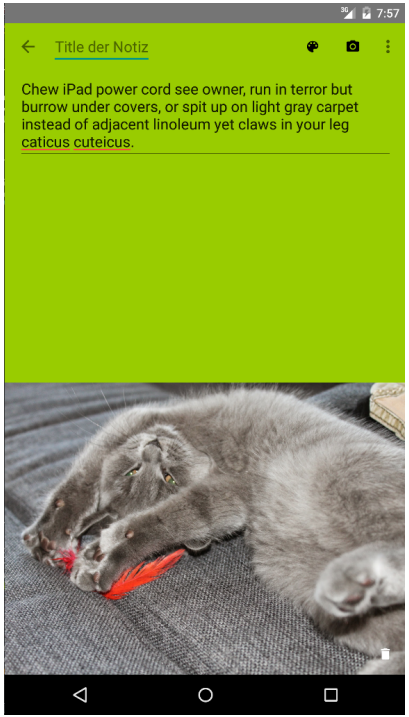
...

Eine Notitz App

2 Ansichten entwickelt:

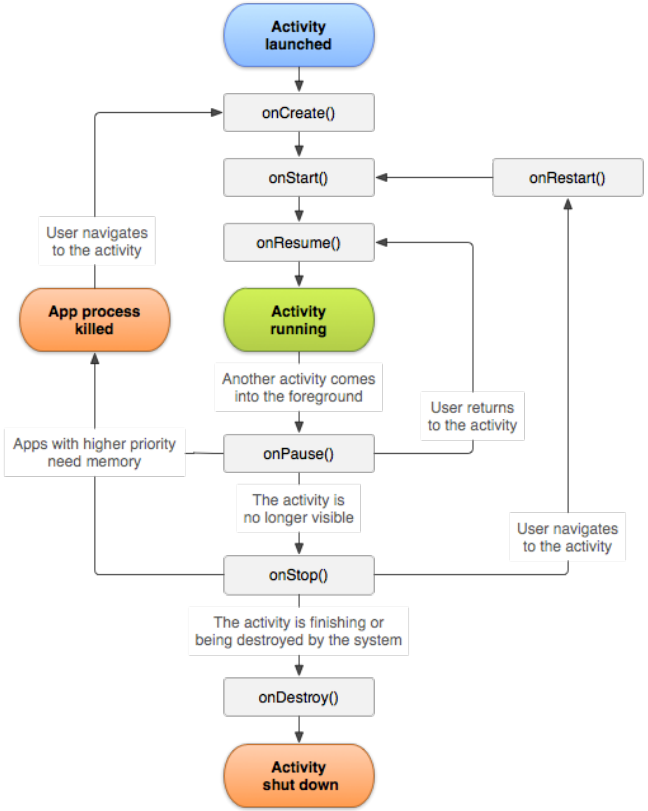
1. Übersicht (Anlegen, Details anzeigen / bearbeiten)
2. Detailansicht (Bearbeiten): Titel, Beschreibung, Bild

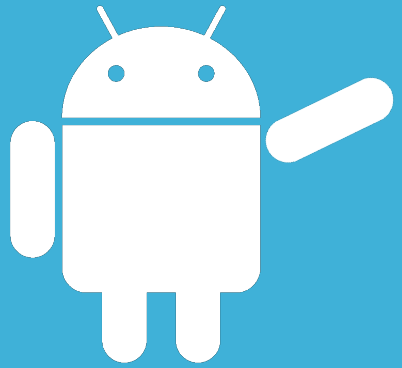




Eine Notitz App

- Activity und Fragment als “View Controller” mit eigenen Lebenszyklen kennen gelernt.
- Interaction zwischen diesen durch Intents und Interfaces.
- Entkopplung von Activities durch Kapselung der Navigation.
- Modellzustand in einem Store, nicht in der View. Views kennen nur IDs um Zustände zu laden, aber nicht zu verwalten.





PROBLEME???

So weit, so gut

3 Ansichten entwickelt, etwas Logik, für die Datenhaltung und die Interaktion der Komponenten … in 4 Tagen entwickelt.

Die App ist noch sehr einfach gehalten und besitzt kaum Funktionalität.

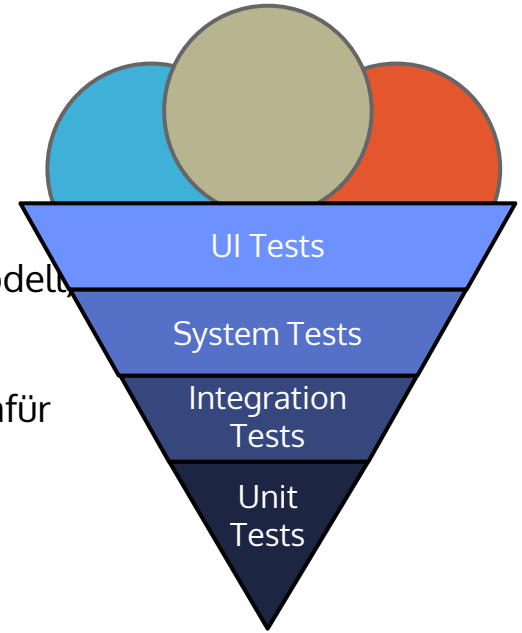
Was wäre wenn …

- Suche / Filter in der Übersicht
- Export von Notizen
- Teilen einer Notitz
- Archivieren von Notizen
- Einbindung in Automationsapps (If This Then That)
- Ausdrucken einer Notiz
- kollaboratives Arbeiten an einer Notiz
- …

Dann …

- Mehr und komplexere Activitys, Fragments: längere Lifecycle-Methoden, Überschneidung im Code (Wiederverwendung von Methoden)
- Wir brauchen Tests, um das steigende Fehlerpotential zu behandeln und Regressionsfähigkeit herzustellen.

Testen unter Android :-(



Probleme mit Activities

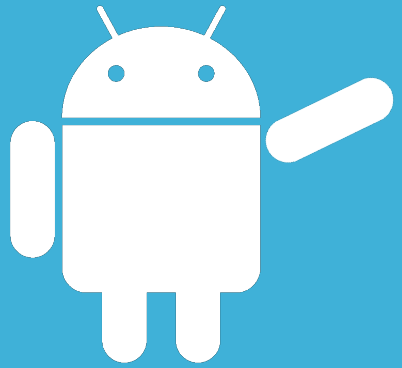
Views mit Controller-Logik und Lifecycle (komplexes Zustandsmodell)
→ deshalb schlecht testbar.

Google selbst sieht Instrumentationtests (UI-Systemtests) vor. Dafür wird die komplette Android-Umgebung bereitgestellt.

- Das ist langsam, umständlich und ich teste zu viel UI mit Businesslogik (Testpyramide → Test-Eistüte)
- Testgetriebe Entwicklung ist damit unmöglich.

Die Android Studio - Testing Odyssey: http://www.openkeychain.org/odyssey-with-testing/


Was ist ein System ohne Tests?
... ein Legacy System

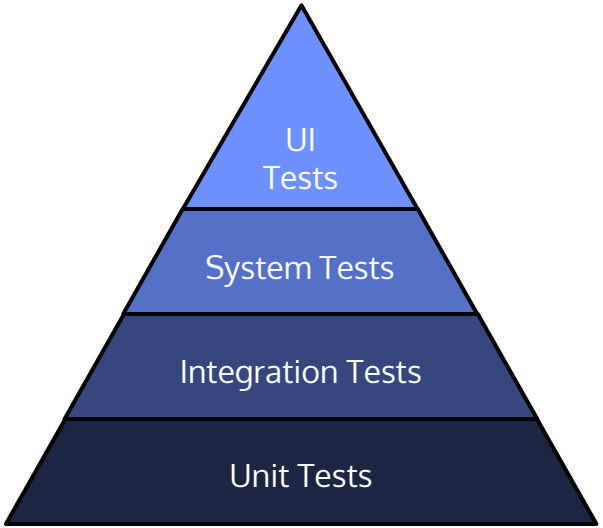


ALS ANDROID-ENTWICKLER WÜNSCHE ICH MIR

eine testbare und flexible Architektur

Saxonia Systems
So geht Software.

Ziele meiner Architektur



Testbarkeit:

Bis zu 100% der Applikationslogik kann getestet werden.

So viele Tests wie möglich sollen Unit Tests sein.

Sicherheit bei Veränderungen (Regressionsfähigkeit).

Flexibilität: Wartbarkeit und Erweiterbarkeit:

Einfache Änderungen einfach umsetzen können.

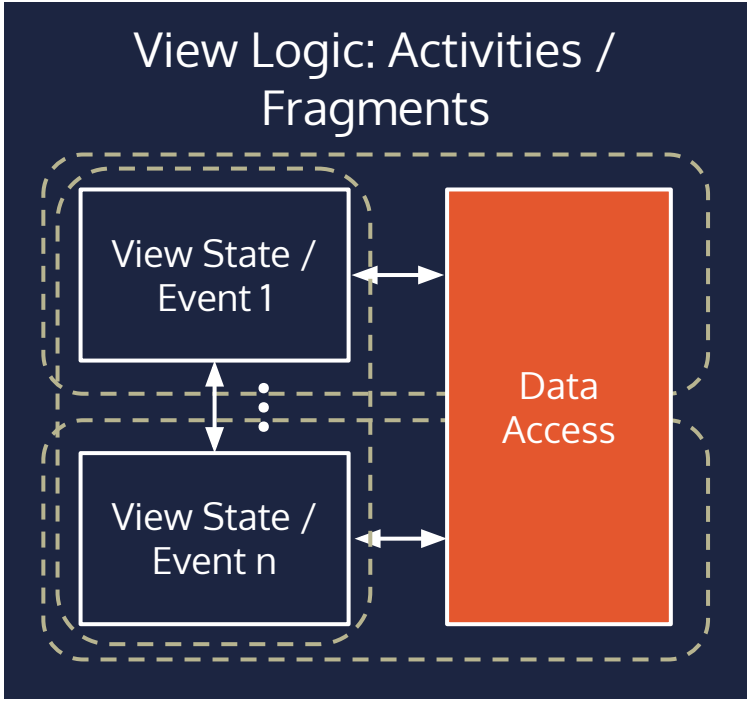
Komplexe Änderungen sind aufwändig aber mit kontrolliertem Risiko machbar. (kontrolliert durch Tests)

Testbare und flexible Architektur - Wie kann das erreicht werden?

Klingt nach einem Fall für Patterns,
einer bekannten Geheimzutat ...

und ein bisschen Frameworks :-)

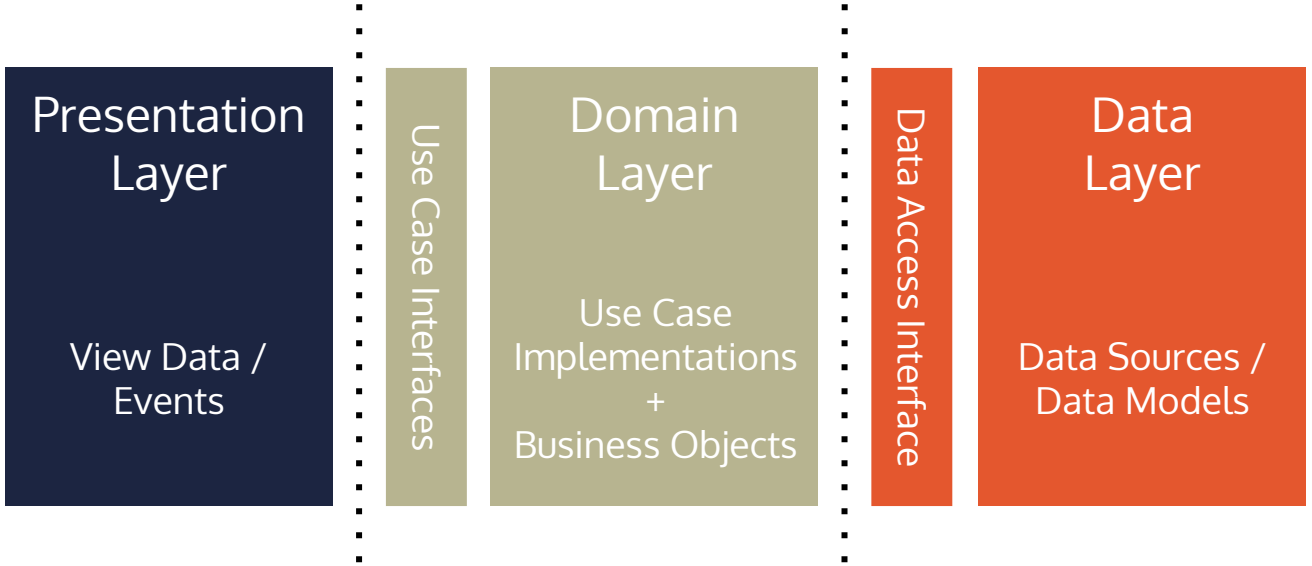
Aktuelle Architektur: Wo sind unsere Business Rules?



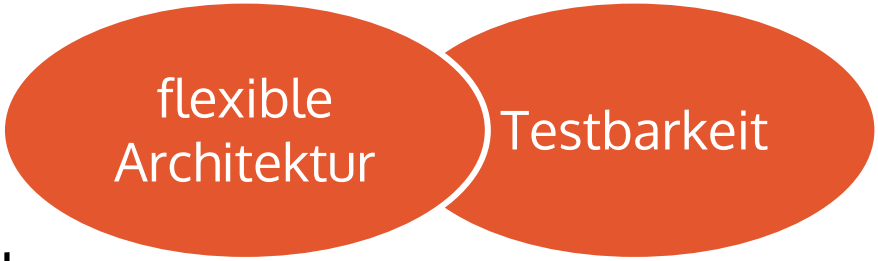
Die bekannte Geheimzutat?
Löffel auf …

Separation of Concerns?!
Schichtenarchitektur.

Abstrakte Schichtenarchitektur



Patterns für unsere Ziele



Dependency
Injection

Command

Entity
Boundary
Interactor

Model View
Presenter

Message
Broker

Publish
Subscribe

Repository

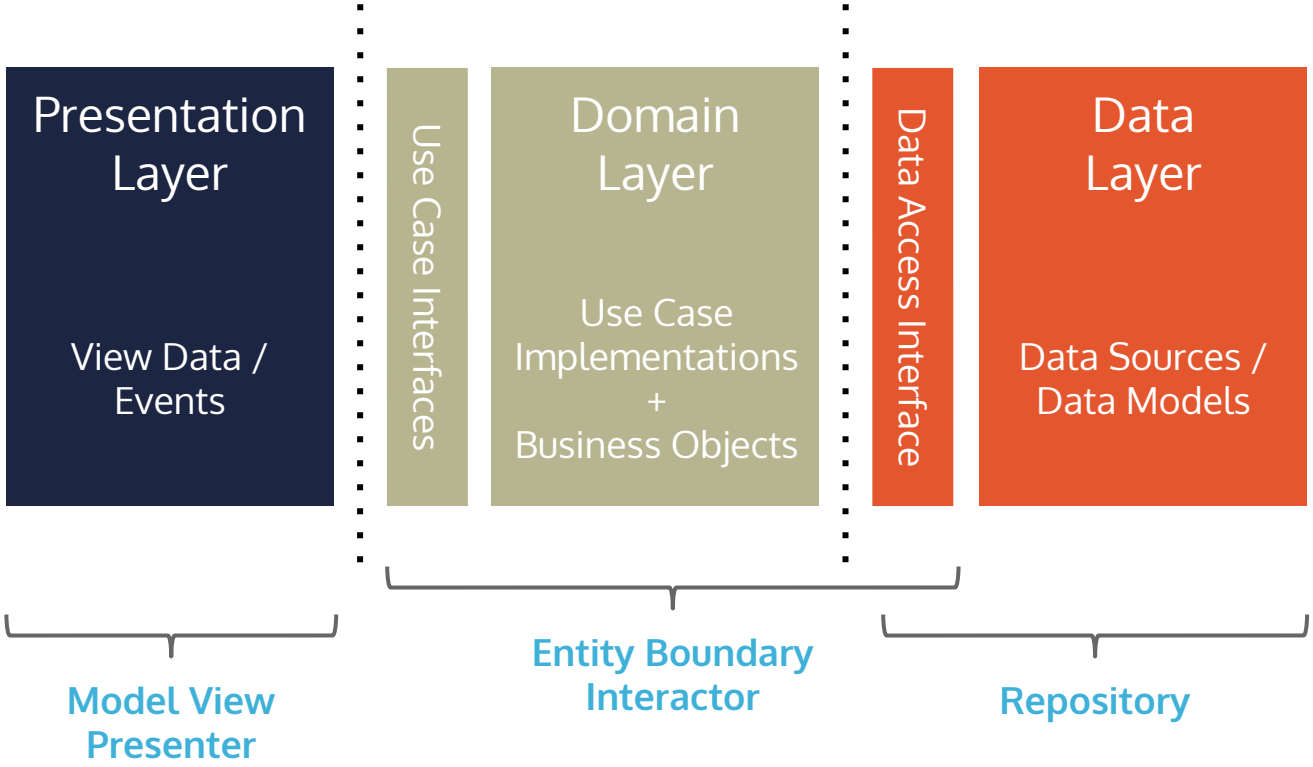
Patterns - inhärent in Android

Memento (View State Saving: Bunde = Memento)

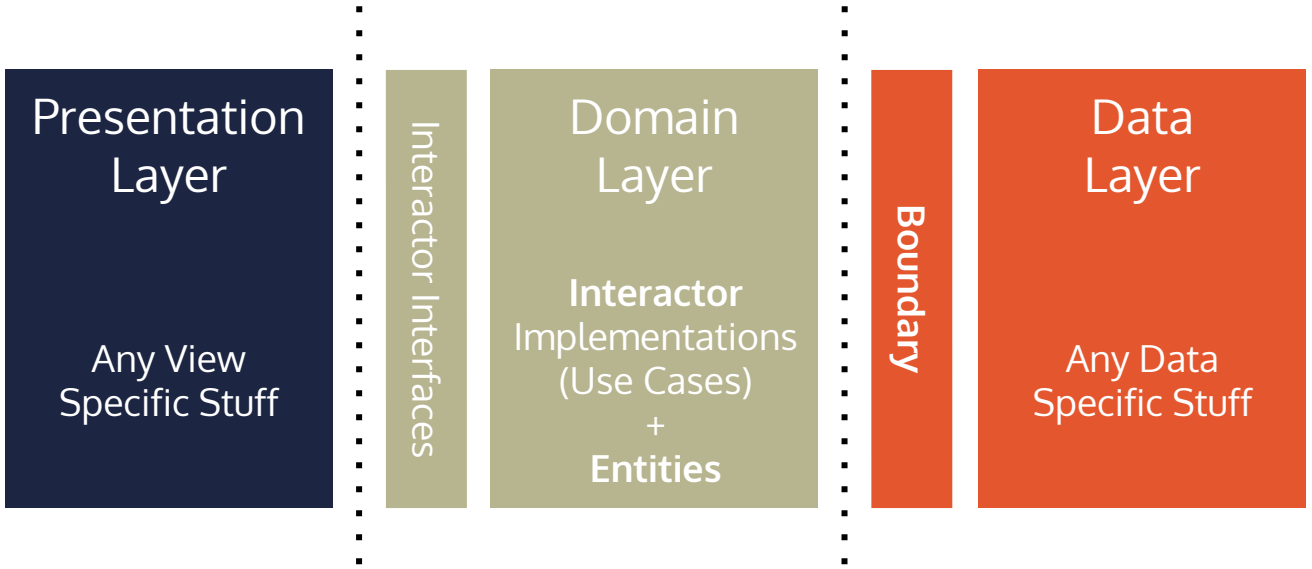
Chain Of Responsibility (Intents und Intentfilters)

Adapter: abstrahiert Entitäten und ihre Beschaffung und Darstellung für einen View-Container

Konkrete Schichtenarchitektur



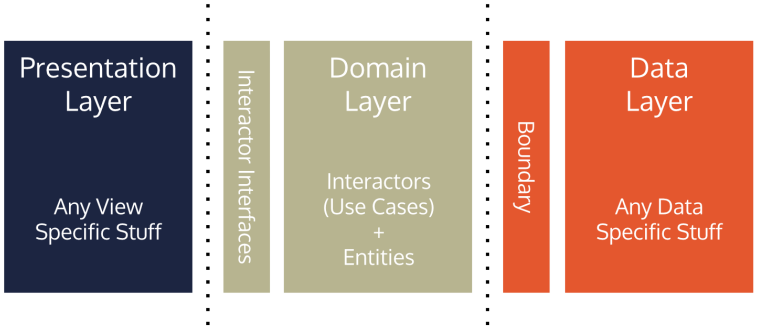
Entity Boundary Interactor (EBI)



Was ist der Mehrwert von EBI

→ Separation of Concerns

Meine Business Rules sind genau definiert. Ich kann sehen / testen, welche Business Rules (Funktionalität) meine Anwendung besitzt.



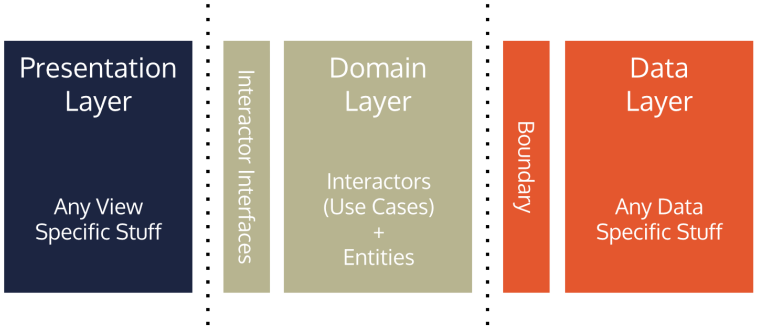
Implementierungsdetails wie die View-Repräsentation (Aufbereitung etc.) und die Datenhaltung (Beschaffung, Mapping, etc.) werden entkoppelt von meinen Funktionalitäten (Business Rules).

Das macht sie nicht nur expliziter erkennbar, sondern auch testbarer.

Was ist der Mehrwert zur Implementierung ohne EBI?

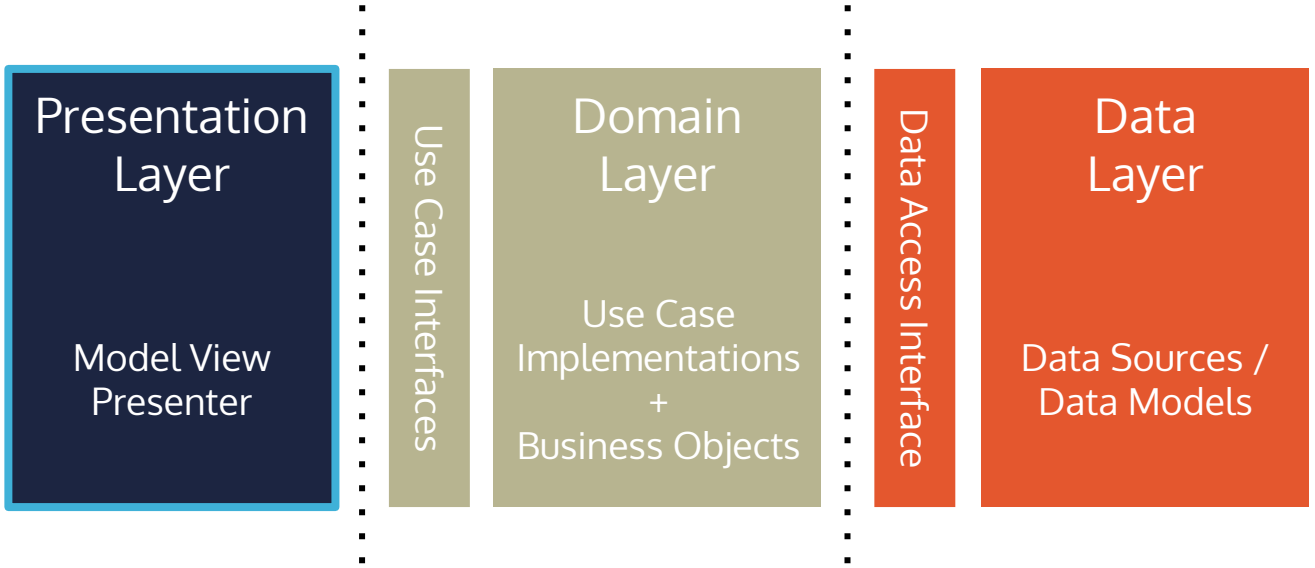
Ich kann auch eine testbare und sinnvolle Architektur ohne EBI erreichen.

Möglicherweise sind meine Business Rules auch gut erkennbar und unabhängig von Implementierungsdetails.

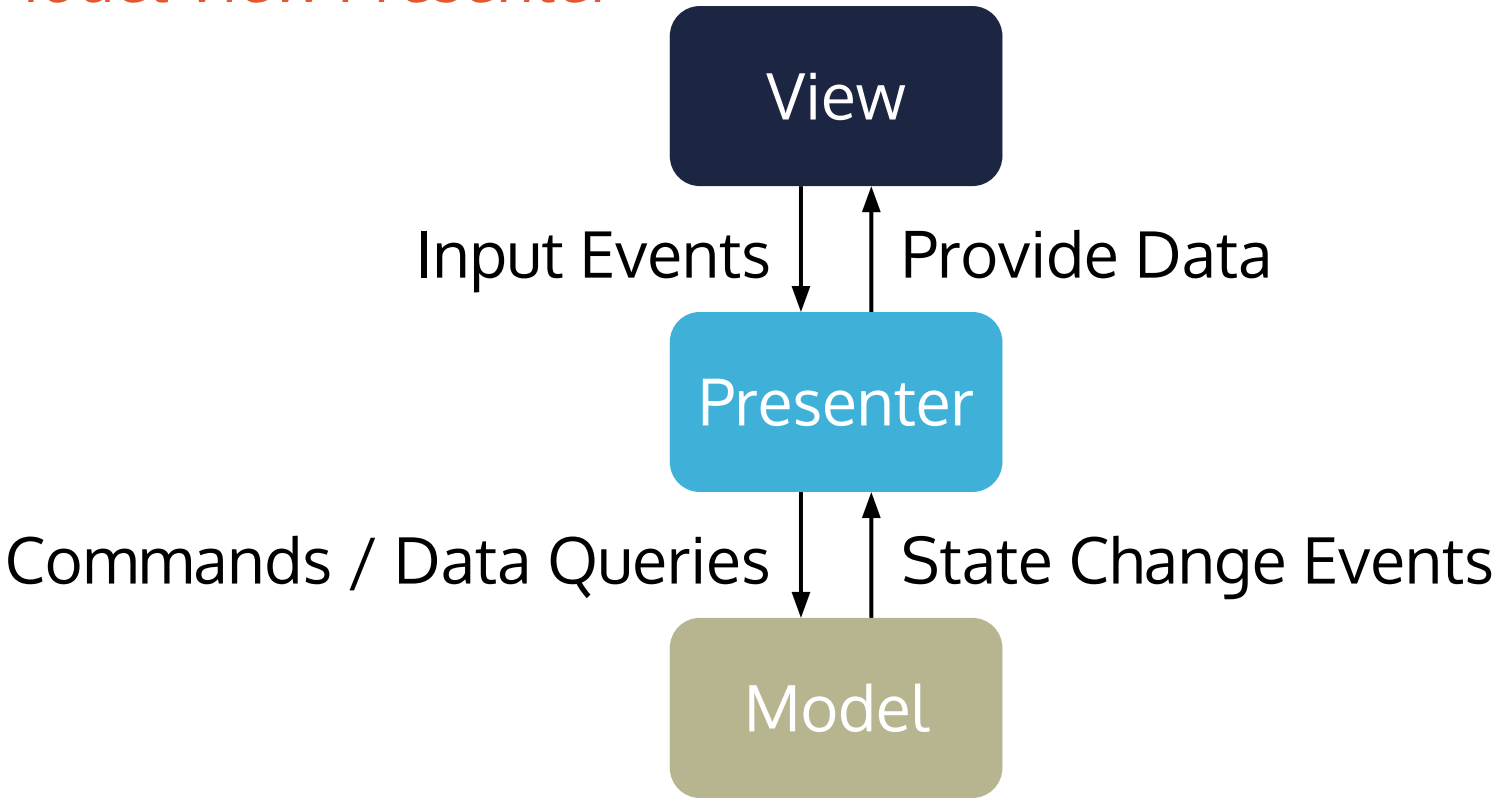


Aber das ist nicht sicher. Das EBI Pattern setzt dies durch und macht meine Business Rules auf jeden Fall explizit und unabhängig testbar.

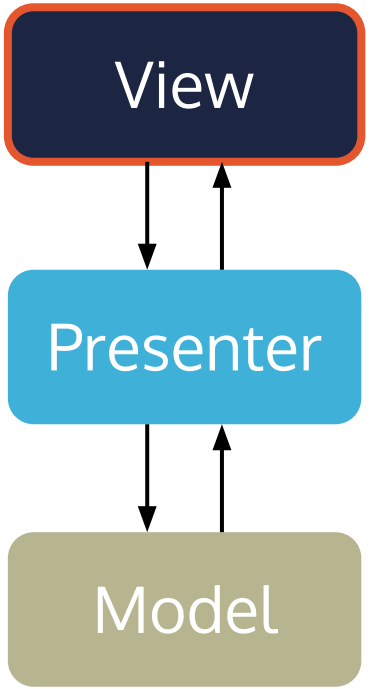
Model View Presenter



Model View Presenter



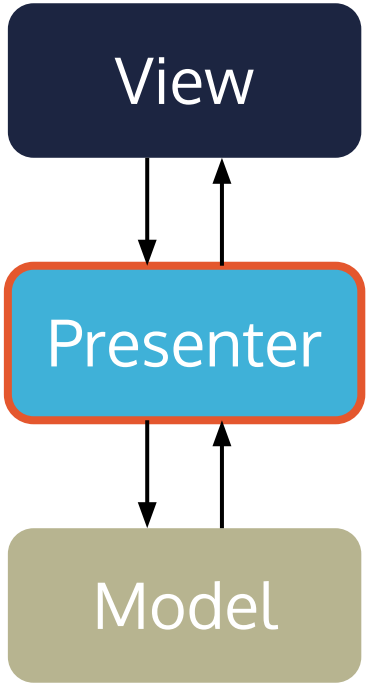
Model View Presenter



- Die View ist eine Activity, ein Fragment oder eine Custom View.
- Sie implementiert ein View-Interface, sodass sie mit einem Presenter kommunizieren kann.
- Sie produziert Input-Events und leitet sie an den Presenter weiter. (Delegation)
- Events aus der Außenwelt (Model) werden nicht durch die View wahrgenommen. Sie kennt nichts außer den Presenter.
- Der Presenter übergibt der View Anzeigedaten oder ändert ihren Zustand.

Die View enthält also nur Glue-Code, der durch Integrationstests abgetestet werden kann. Sie ist “dumm”

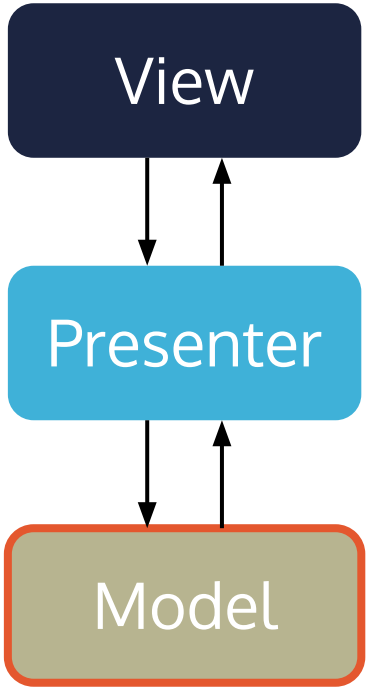
Model View Presenter



- Der Presenter definiert ein View-Interface. Gegen dieses kann er kommunizieren.
- Der Presenter orchestriert Interactors, um View-Events zu verarbeiten und tritt damit indirekt mit dem Data Layer in Kontakt.
- Er empfängt Events (via Callbacks) vom Domain Layer und bringt Daten ggf. in Präsentationsform, sodass diese an die View zur Anzeige gereicht werden können.

Der Presenter verwaltet also das Zustandsmodell der View. Daher kennt er den Lifecycle der View.

Model View Presenter

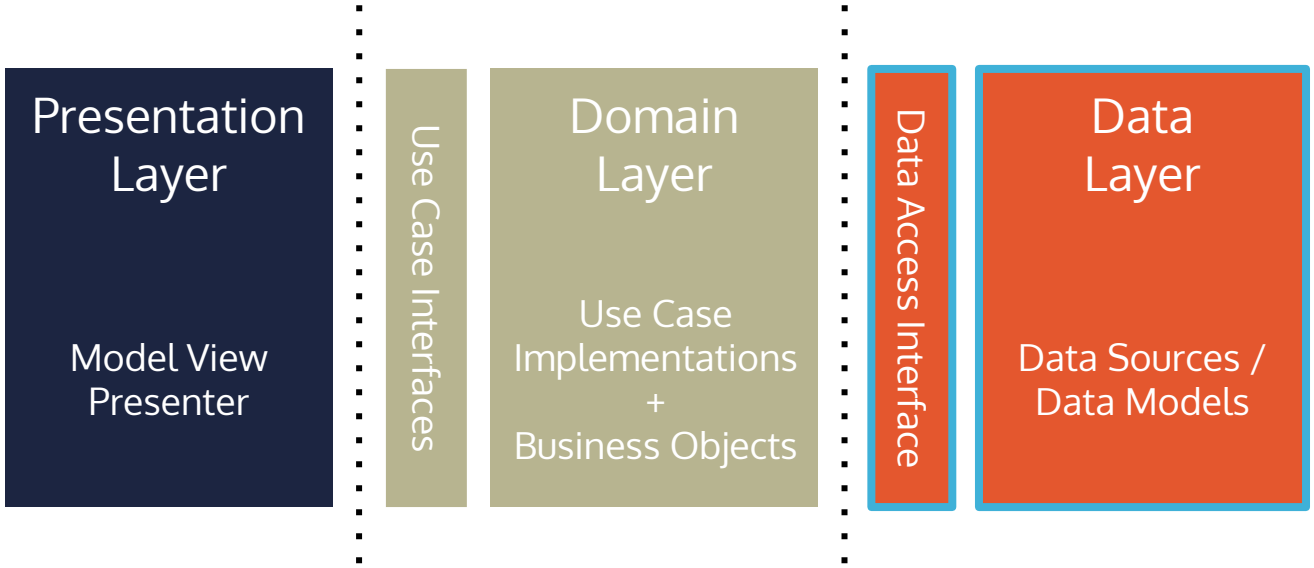


Das Model ist nicht das Model des Domain Layers, sondern ein eigenes für den Presentation Layer spezifisches Model.

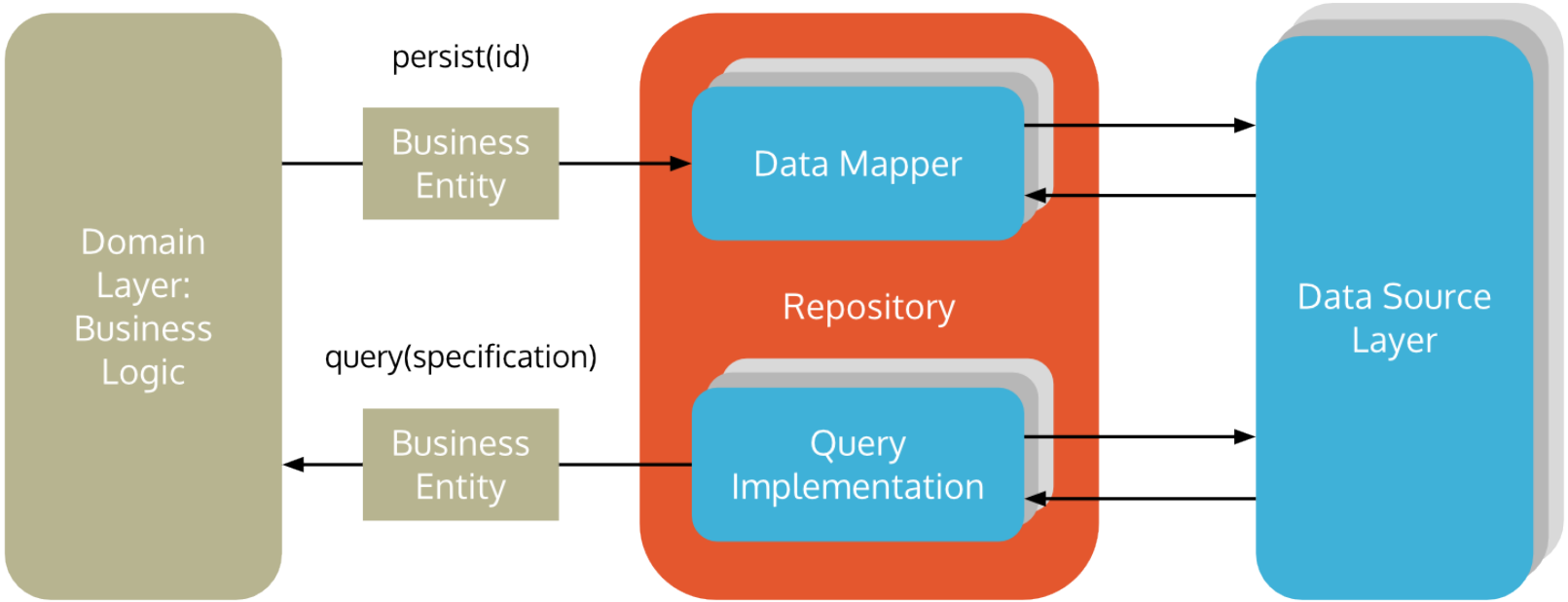
KISS: Es kann jedoch auch das Model aus dem Domain Layer benutzt werden

Das Model wird effektiv durch den Domain Layer bereitgestellt. Der Begriff Model heißt nicht nur POJOs sondern auch Business Objects - diese Enthalten Logik zum Laden der Model-Daten.

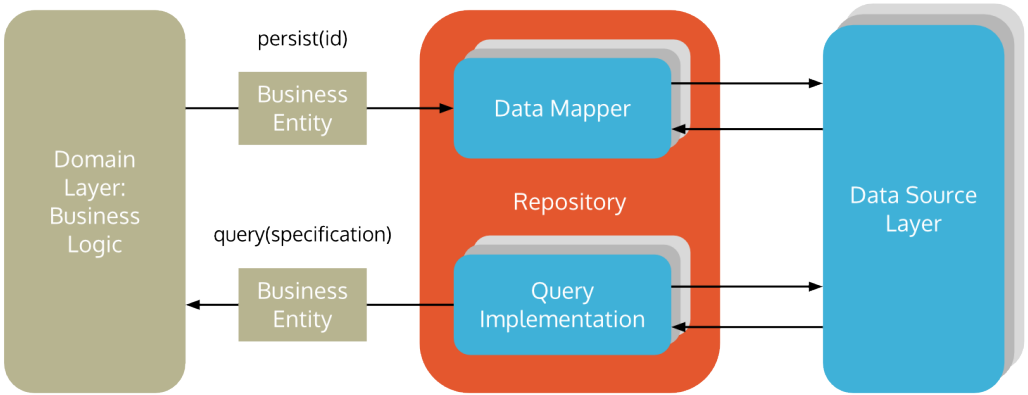
Repository


Presentation Layer
Model View Presenter
Use Case Interfaces
Domain Layer
Use Case Implementations
+
Business Objects
Data Access Interface
Data Layer
Data Sources / Data Models

Repository


Domain Layer: Business Logic
persist(id)
Business Entity
query(specification)
Business Entity
Data Mapper
Repository
Query Implementation
Data Source Layer

Repository



- Businesslogik weis nicht woher Daten kommen.
- Datenbeschaffungslogik leicht austauschbar - auch zur Laufzeit (Strategy Pattern).
- Übersetzung unterschiedlicher Datenrepräsentationen.
- Query-Logik zentral, Redundanz besser vermeidbar.
- Beim Austausch der Datenbeschaffungslogik bleibt die Businesslogik unverändert.

Repository implementieren

Unterschiedliche Datenzugriffsimplementierungen:

- Webservice-Aufrufe in die Cloud: diverese JSON oder XML-Mapping Frameworks
- ORM auf Sqlite: ActiveAndroid, greenDAO
- Manuelle Persistenz in einer Datei oder Sqlite-Datenbank

Message Broker + Publish Subscribe: Event Bus

Events bzw. ein Event Bus kann helfen, Applikationsteile (Komponenten) zu entkoppeln.

Events sind nicht mehr als Benachrichtigung über Zustandsänderungen, keine Kommandos oder Anweisungen!

Zuammangehörige Klassen sollten direkt miteinander kommunizieren. Es geht um die Frage, ist es wichtig oder hilfreich für das Verständnis, wer für eine Aktion verantwortlich ist?

Falls nicht, sind Events durchaus eine Möglichkeit Abhängigkeiten zwischen Komponenten zu reduzieren.

Für Android sind diverse lightweight Event-Bus-Frameworks zu finden: GreenRobot's EventBus, Square's Otto

Dependency Injection

```
public class ToTest {
  String getCustomerName(long id) {
    new DatabaseConnection().query(
      "SELECT name FROM
      customers WHERE id = " + id) ;
      ...
      return foundName;
  }
}
```

Frameworks: Dagger, Butter Knife

```
public class ToTest {
@Inject
private DatabaseConnection dbConn;
  String getCustomerName(long id) {
    dbConn.query("SELECT name FROM
      customers WHERE id = " + id) ;
      ...
      return foundName;
  }
}
```

Testen

Um Tests unter src/test/java in der IDE angezeigt zu bekommen und diese ausführen zu können muss man:

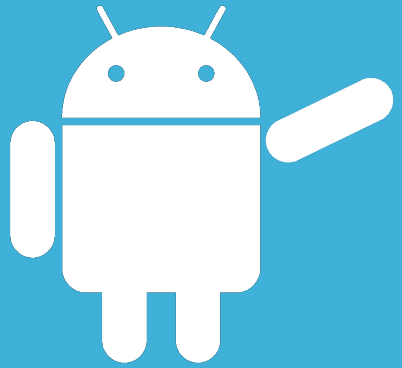
- den “Unit Testing support” aktivieren (je nach Version des Android Studio)
- und die Ansicht in “Build Variants” auf “Unit Tests” setzen.

http://tools.android.com/tech-docs/unit-testing-support

Mocking der Android-Umgebung: Robolectric, Robotium ...

Was macht Robolectric?

Stellt die Android-Umgebung innerhalb der JVM bereit und erlaubt so Tests in der IDE auszuführen, ohne ein Deployment auf einem Emulator ausführen zu müssen.



onFinished()

Heute was gelernt?

Ohne Tests für meine App-Logik entwickelt sich Legacy Code.

Patterns und Frameworks können helfen, testbare und saubere Architekturen zu schaffen. Patterns sind jedoch kein Doktrin - denn sie bringen häufig Komplexität, die abgewogen werden muss.

Abweichungen sind durchaus sinnvoll, wenn:

- Code einfacher und besser verständlicher wird (KISS)
- die Testbarkeit dadurch erhöht, ...

Heute was gelernt?

Kombiniert man unterschiedliche Patterns, ist es nicht immer leicht zu entscheiden, welche Logik, in welcher Schicht oder durch welche Komponenten zu implementieren ist: Validierung in Presenter oder in Interactors / Business Rules (Domain Layer).

MVP ist ohne Framework umsetzbar und ermöglicht SoC. Dadurch wird meine Businesslogik testbar.

UI-Logik- oder Integrationstests jedoch nicht ohne, da das Google-gegebebene Ökosystem hinsichtlich Tests mit JUnit veraltet ist und wenig Hilfen zum Testen auf dieser Ebene bereitstellt. (Stand 05/2015)