Getting clean.

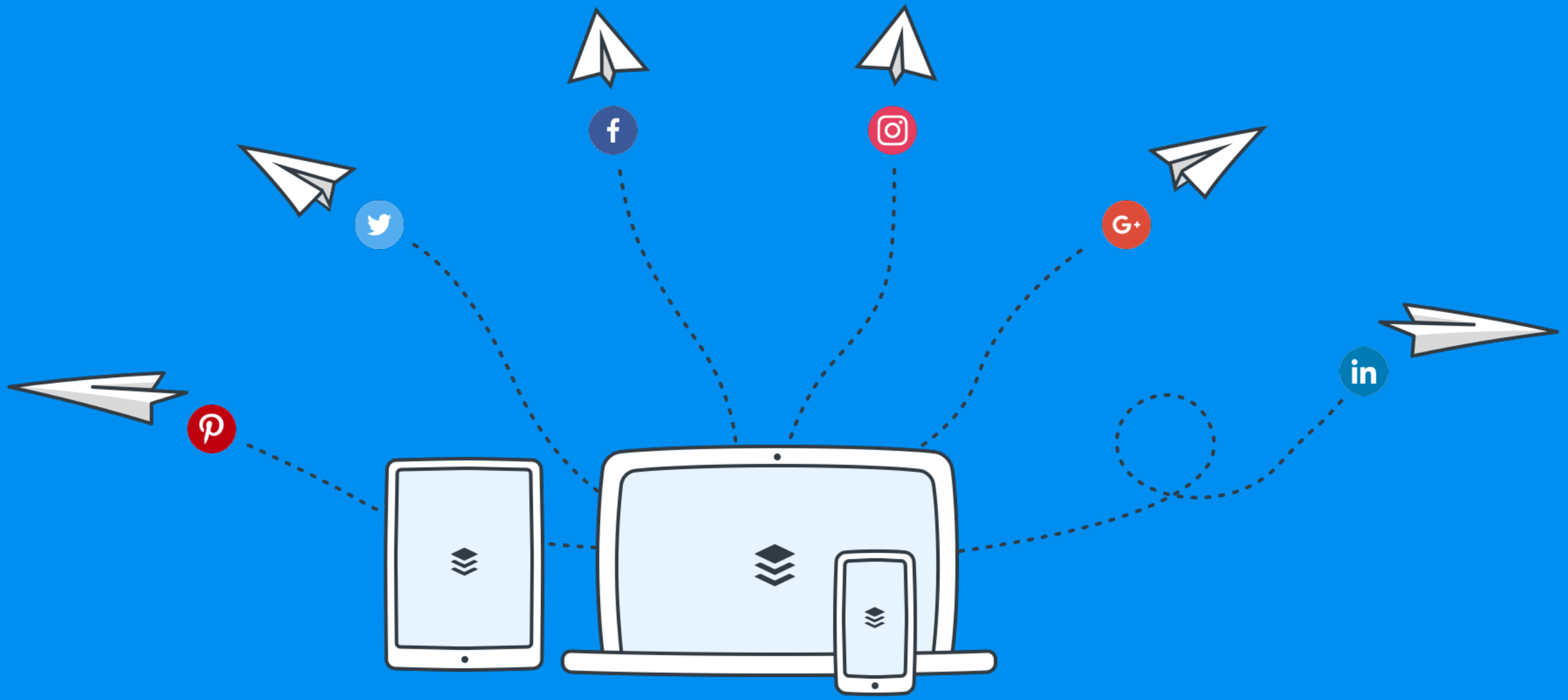
Keeping lean.

Joe Birch - Android Engineer @ Buffer

@hitherejoe

hitherejoe

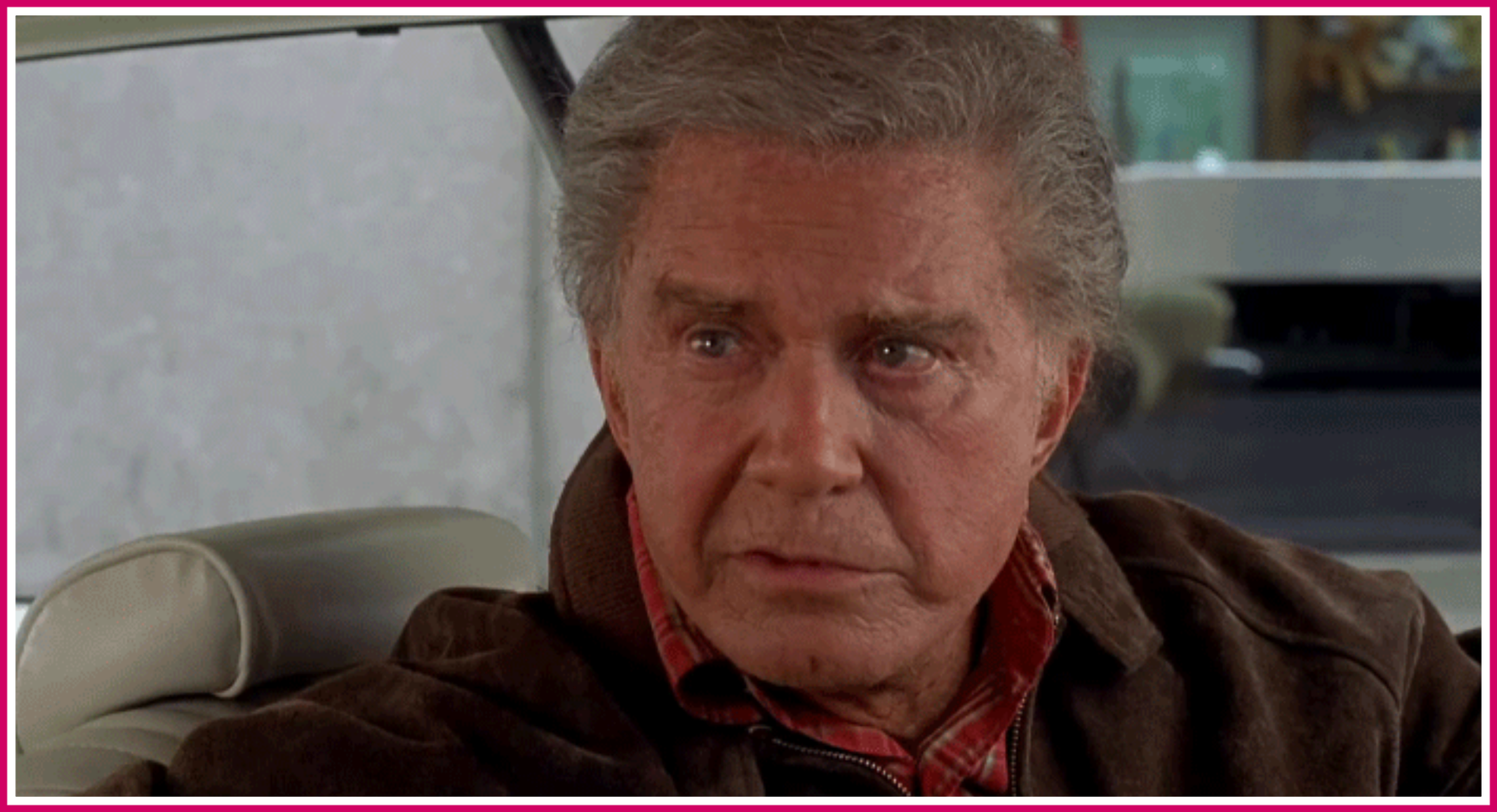
joebirch.co



@buffer

buffer.com

Greenfield projects




SHIP IT

No longer lean

- High-level of technical debt

- Extremely tightly coupled concepts

- No unit or UI tests

- Difficult to know what is going on and where

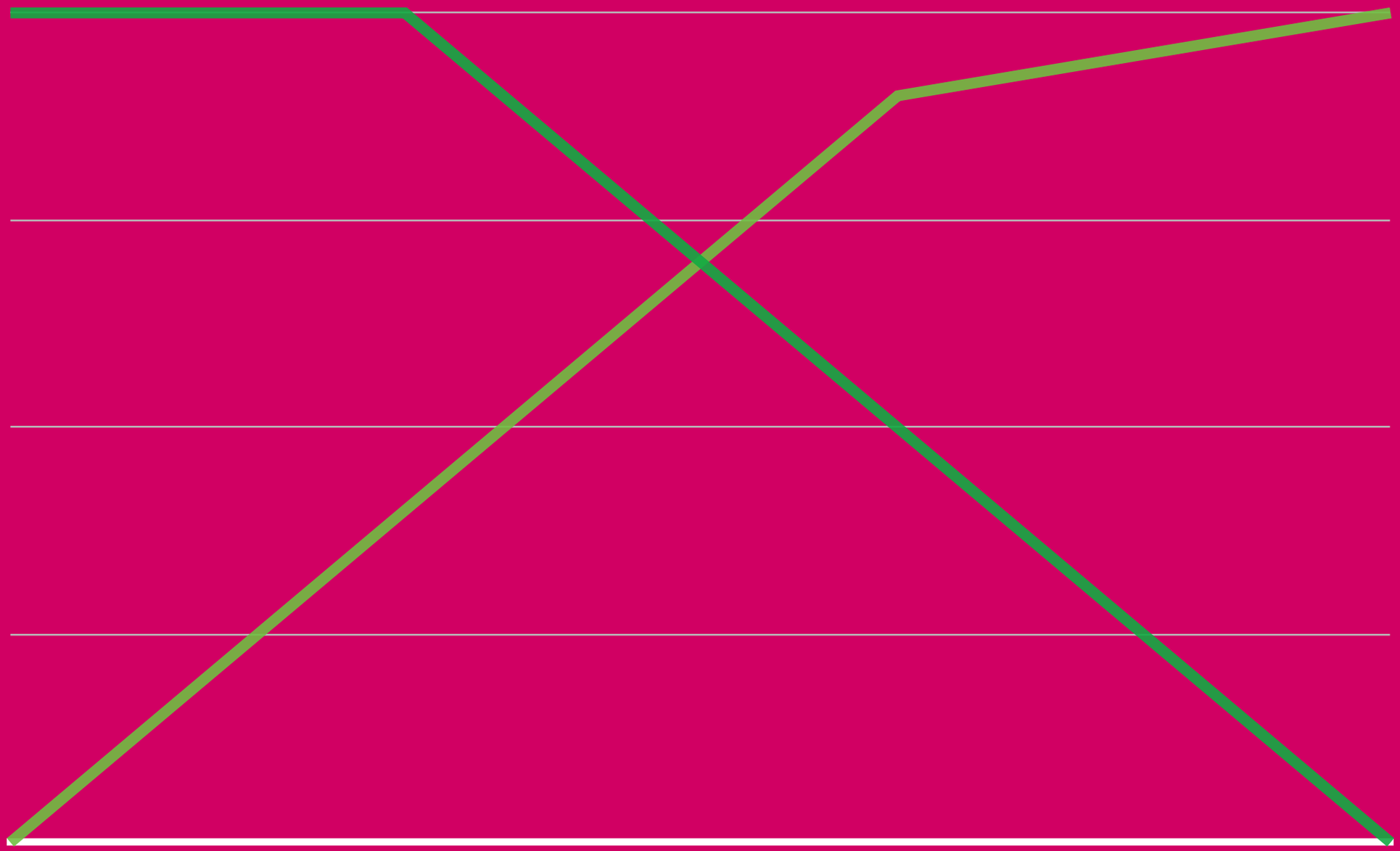
No longer lean

- YOU know what is what (for now)
- YOU know where class X is (for now)
- YOU know where logic for Y is (for now)
- But what about along the line…

Output

Debt

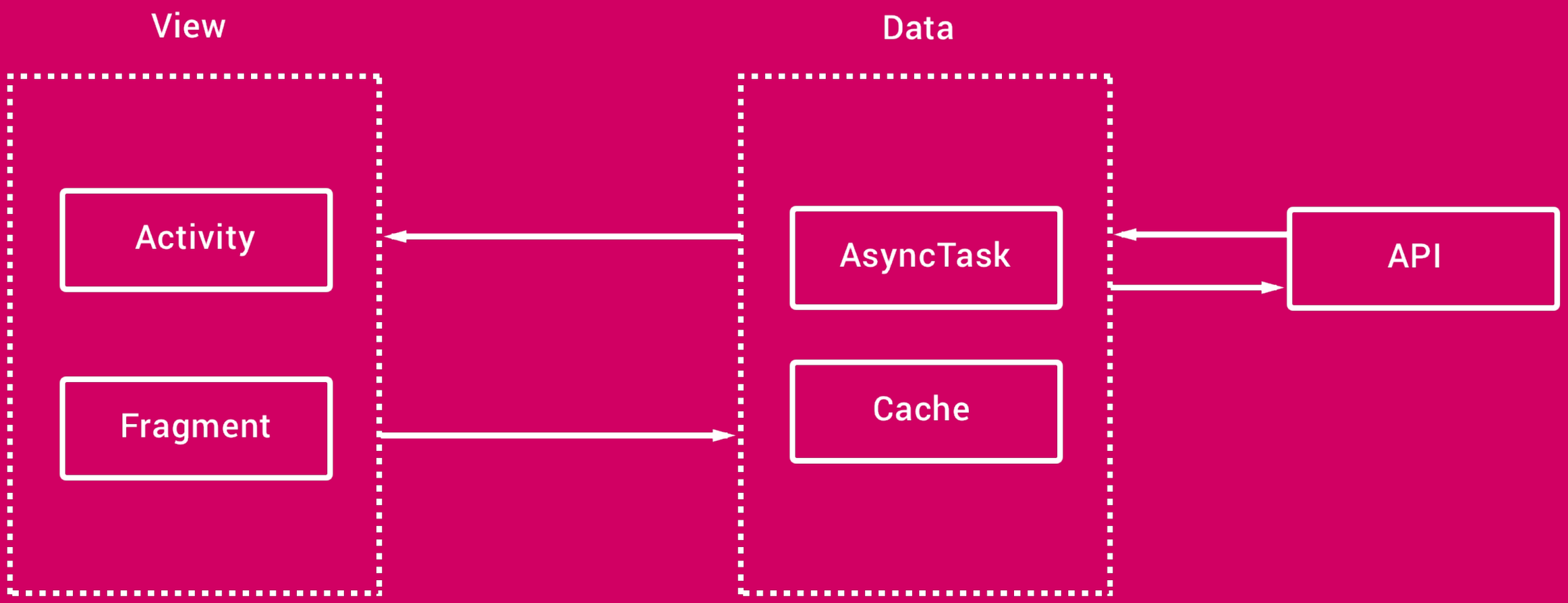
Output

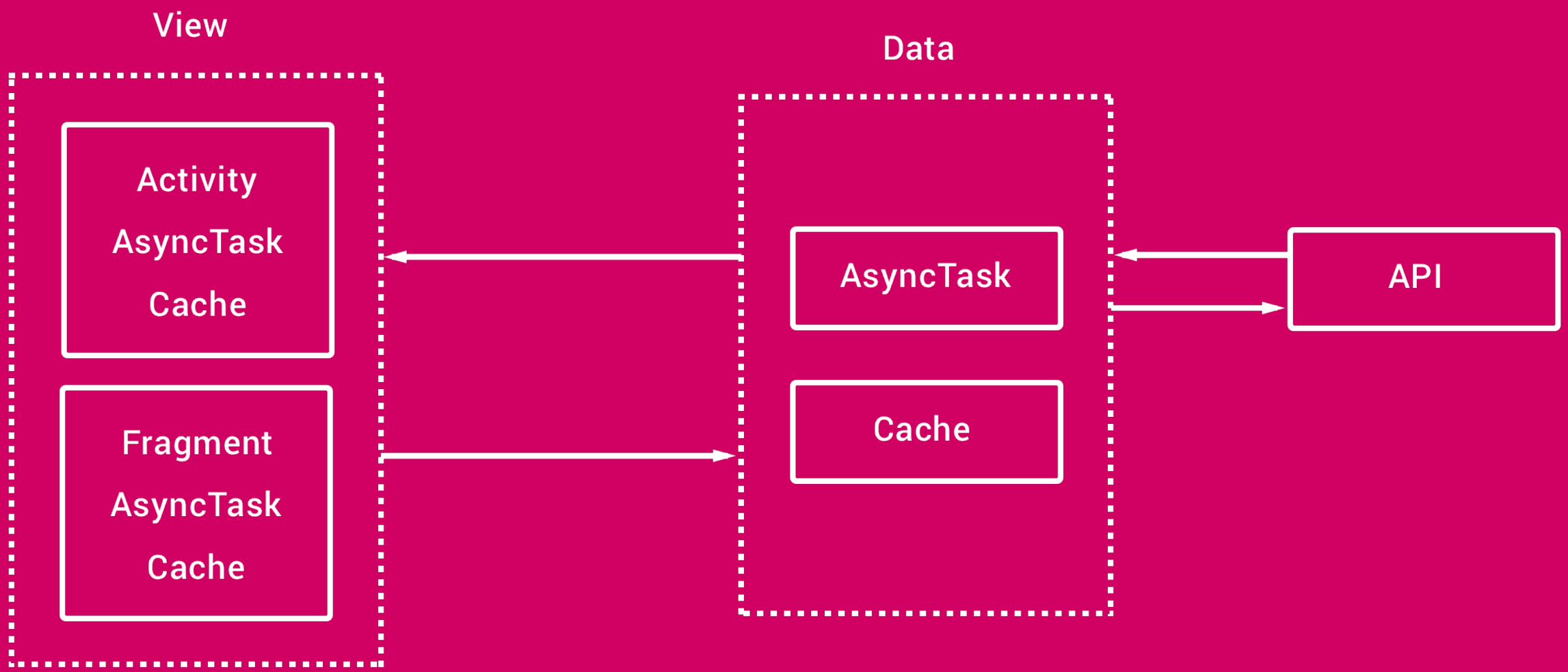


Debt

For the sake of yours and future developers sanity - have empathy. Be mindful of the consequences and think before you craft.

Buffer for Android


View
Activity
Fragment
Data
AsyncTask
Cache
API


View
Activity
AsyncTask
Cache
Fragment
AsyncTask
Cache
Data
AsyncTask
Cache
API

When you delete a block of code that you thought was useless

- Lack of architecture

- Bloated activities with multiple responsibilities

- Difficult to test

- Difficult to extend and maintain

- Duplication of concepts == no reuse

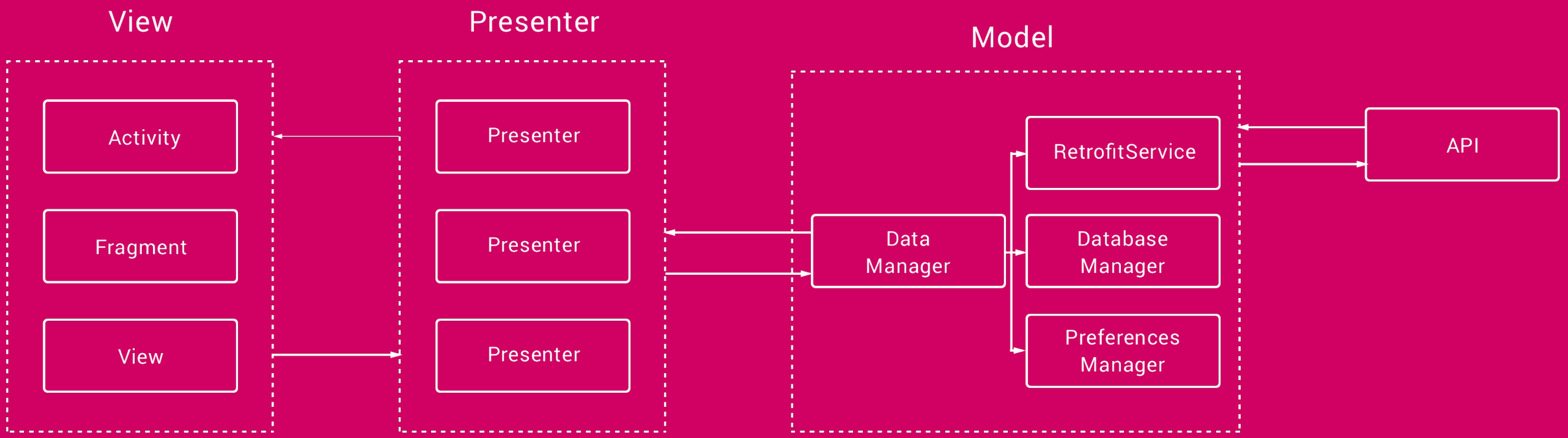
Moving to MVP

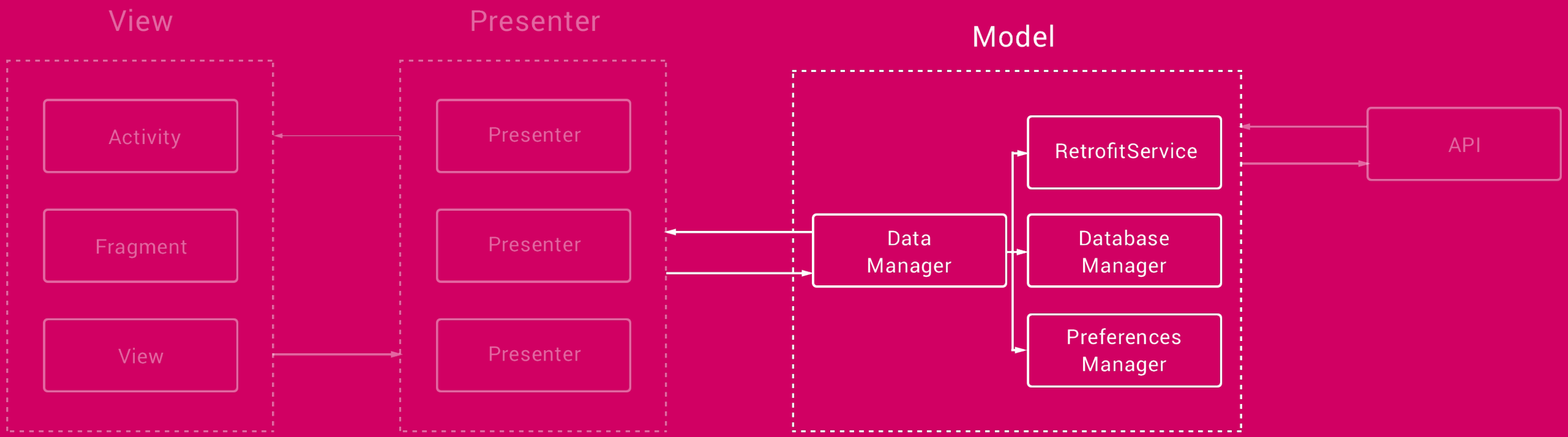
- Fear of changing too much

- Helped separate presentation logic

- DataManagers to house data operations

- Increased test coverage


View
Activity
Fragment
View
Presenter
Presenter
Presenter
Presenter
Model
RetrofitService
Data Manager
Database Manager
Preferences Manager
API


View
Presenter
Model
Activity
Fragment
View
Presenter
Presenter
Presenter
RetrofitService
Data Manager
Database Manager
Preferences Manager
API

Not enough.

- Bloated DataManager classes

- Future changes, offline - where?

- Presentation linked to specific data source

- No clear line between responsibilites

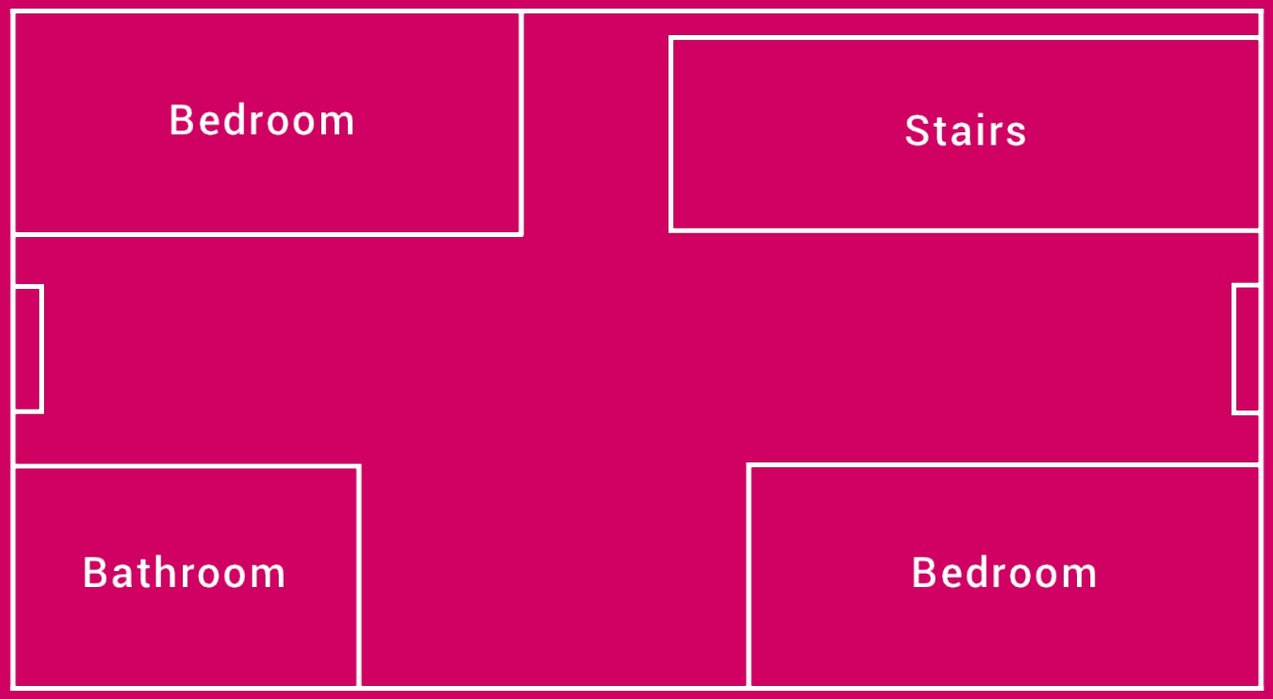
- Buffer is a big app. Data types, features, sources

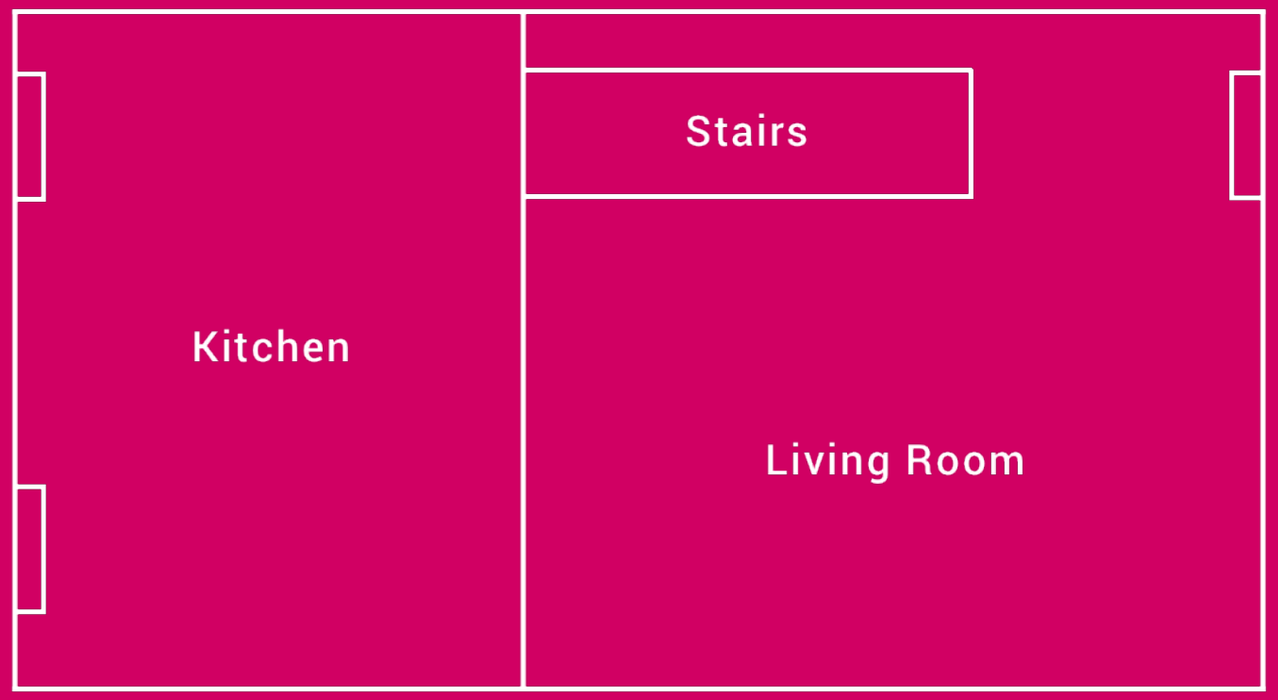
Architecture

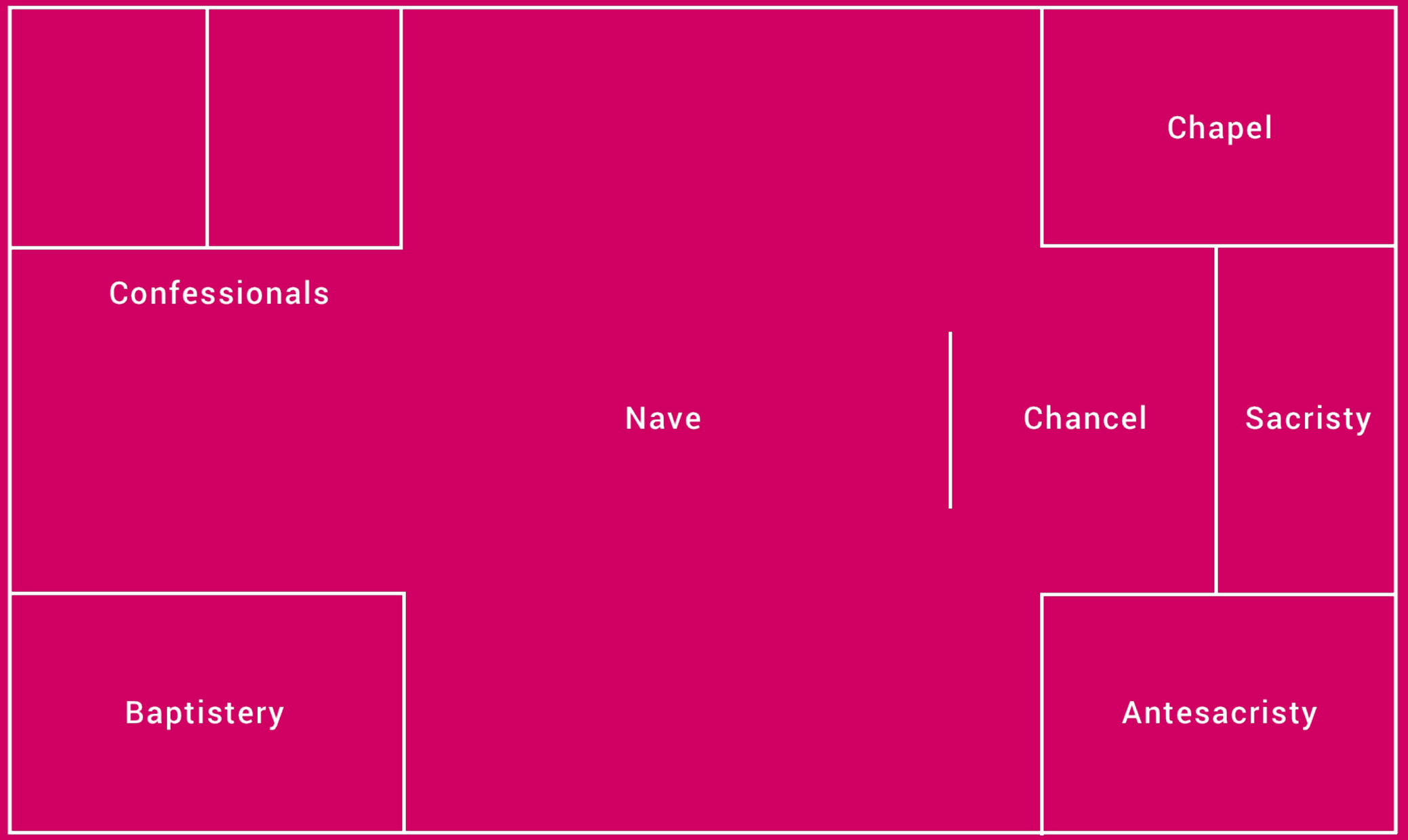
"The art of **planning**, **designing** and **constructing** buildings"

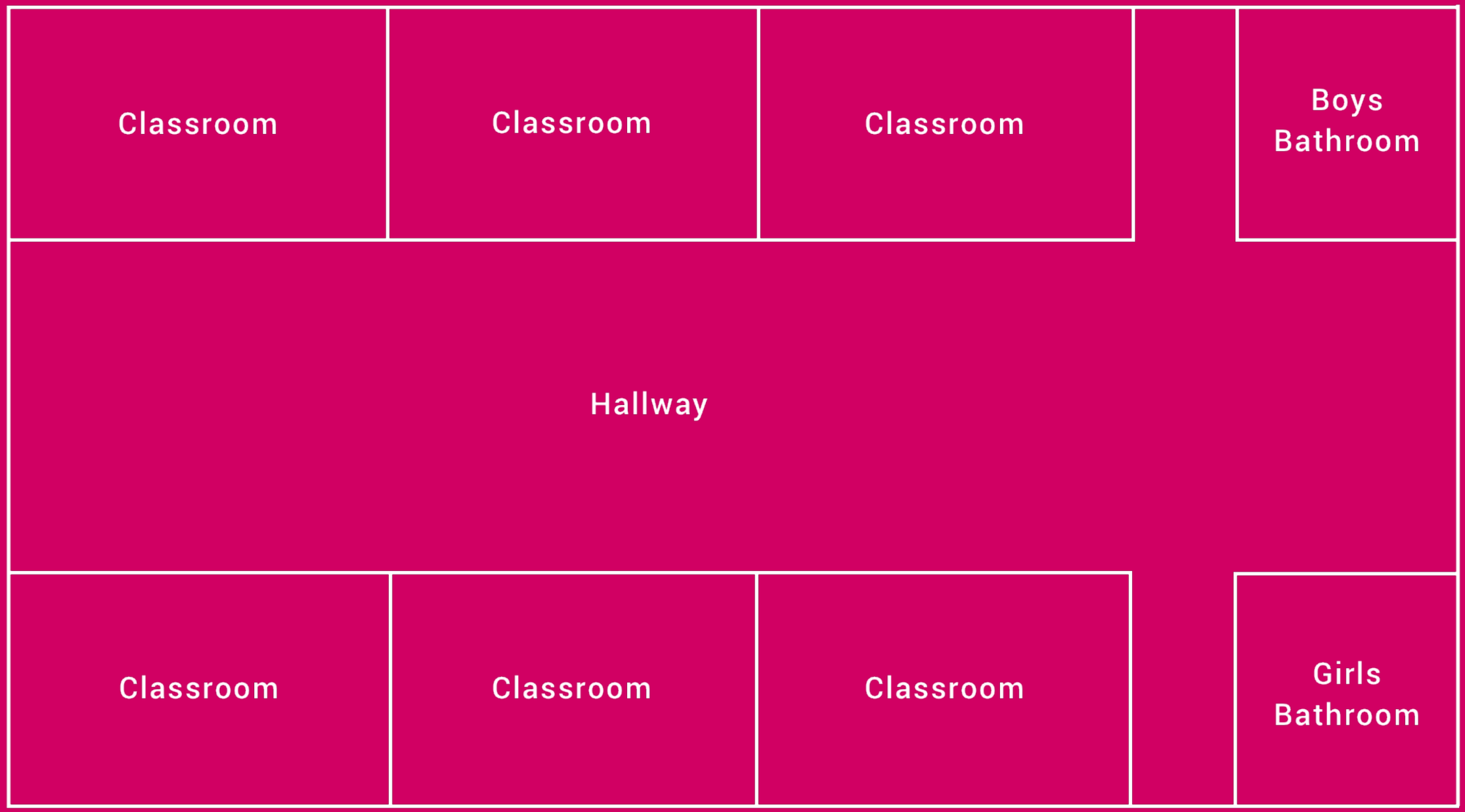
Architecture

"The art of **planning**, **designing** and **constructing** buildings concepts"

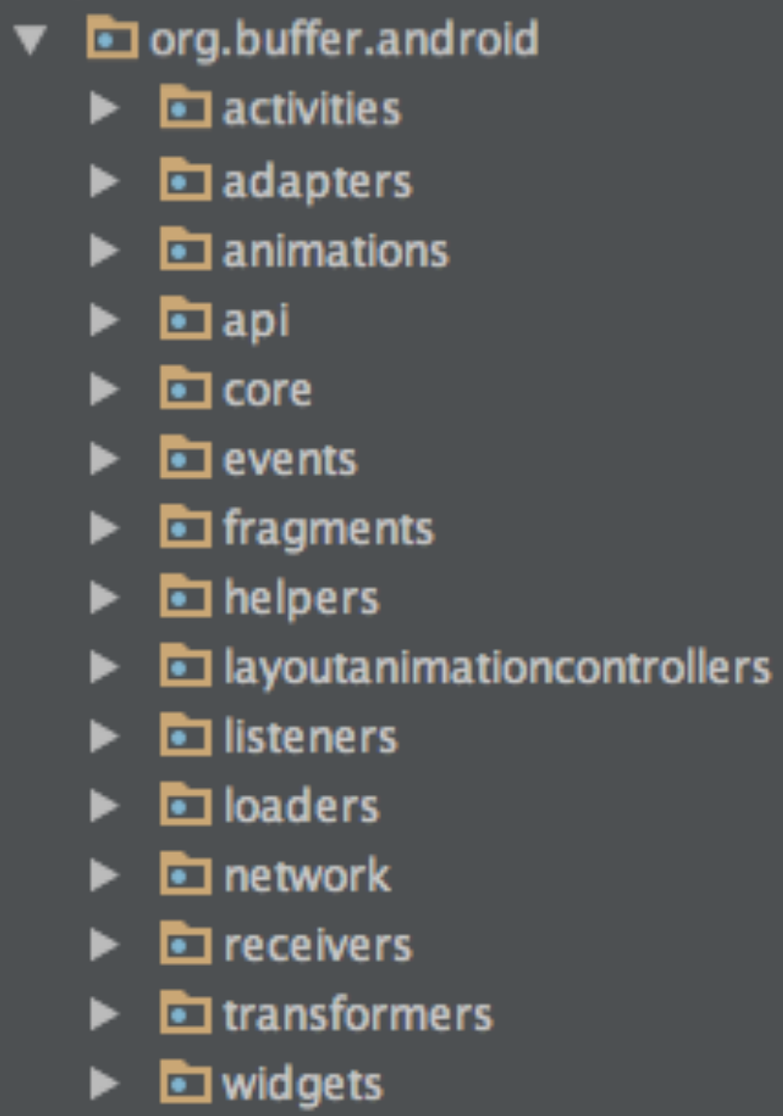

Bedroom
Stairs
Bathroom
Bedroom


Stairs
Kitchen
Living Room


Chapel
Confessionals
Nave
Chancel
Sacristy
Baptistery
Antesacristy

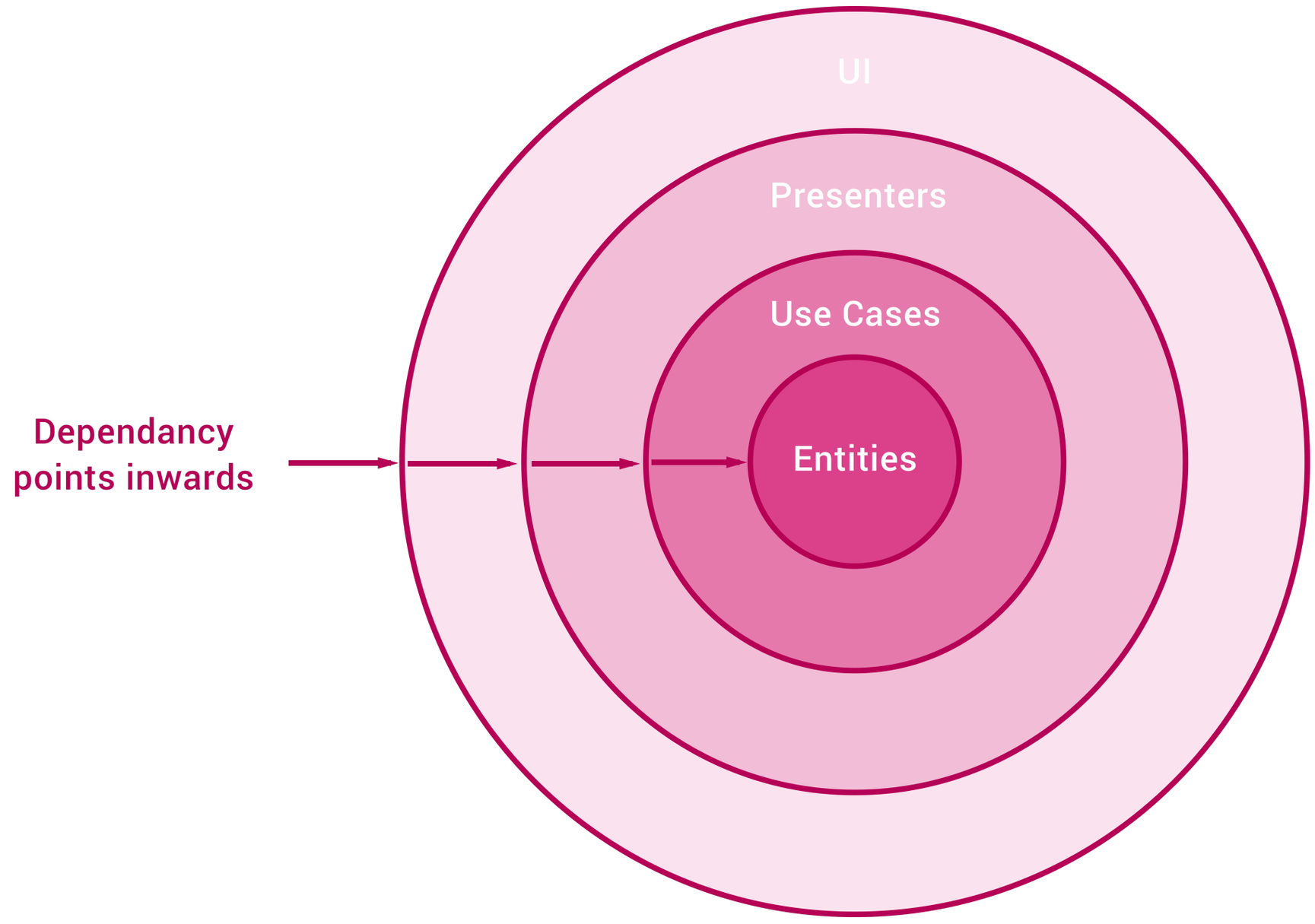

Classroom
Classroom
Classroom
Boys Bathroom
Hallway
Classroom
Classroom
Classroom
Girls Bathroom

activities	fragments	adapters
widgets	network	listeners
receivers	services	animations


org.buffer.android
activities
adapters
animations
api
core
events
fragments
helpers
layoutanimationcontrollers
listeners
loaders
network
receivers
transformers
widgets

Think outside of your team.

Clean Architecture



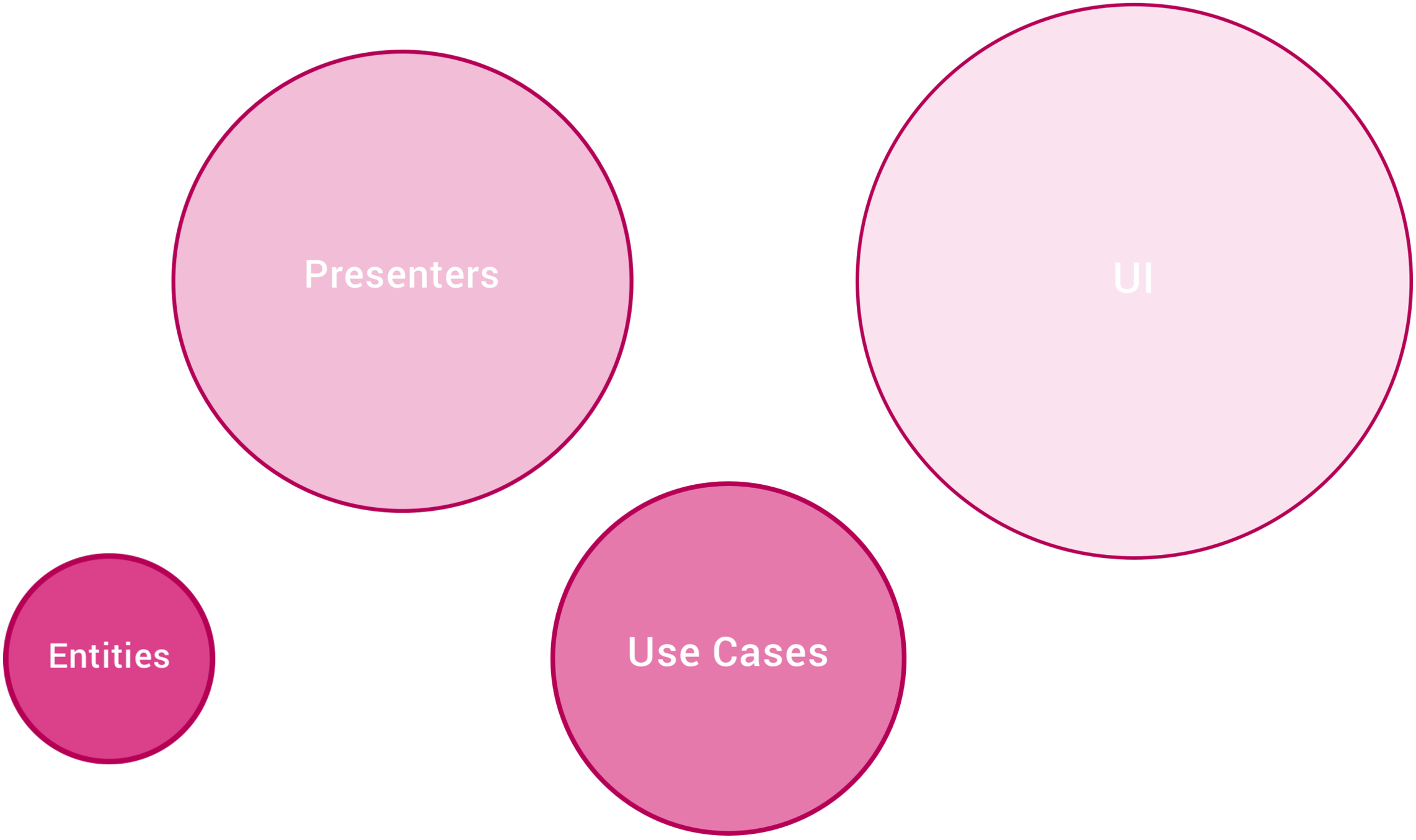
Frameworks and Drivers

Interface Adapters

Business Rules

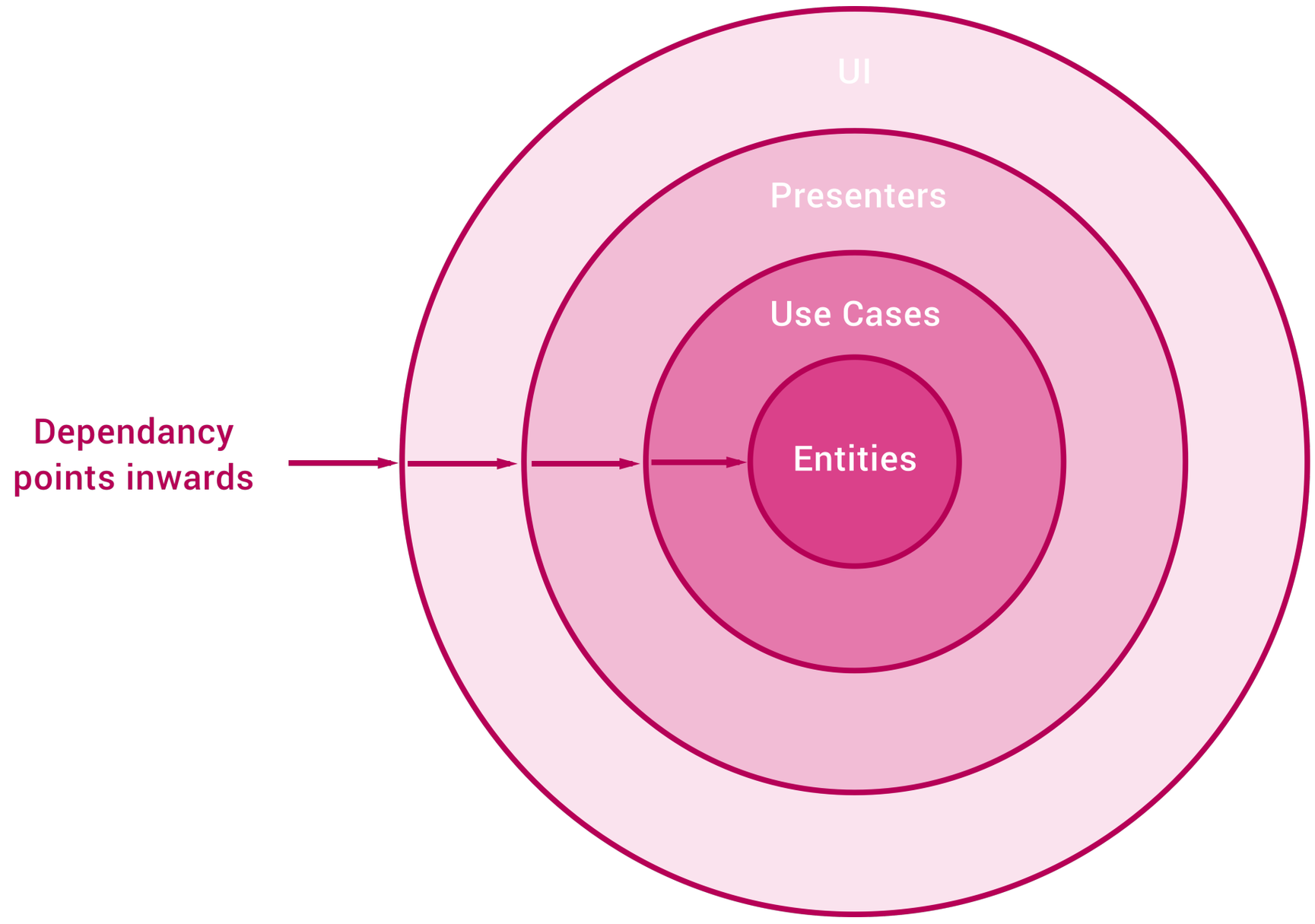
Domain Logic

Separation of Concerns



Separation of Concerns

- Layers are independent of frameworks
- Code becomes more testable
- Inner layers independant of UI
- Inner layers independant of external parties
- Bugs more isolated



Frameworks and Drivers

Interface Adapters

Business Rules

Domain Logic

Layer Models

- Each layer has own Model classes
- Removes dependancy on external models
- Easily map with Mapper classes

Model

Mapper

Model

Mapper

ModelA

Enterprise Business Rules

- Application Business Objects (Entities)
- Unaffected by operational change
- Can create before touching any UI

```java
public class TournamentModel {

    public String id;
    public String name;
    @SerializedName("full_name")
    public String status;
    @SerializedName("date_start")
    public String dateStart;
    @SerializedName("date_end")
    public String dateEnd;
    public int size;

}
```

Application Business Rules

- Application specific business rules

- Uses Cases

- Affected by operational change

- Isolated from external concerns

```
public class GetSchedules extends UseCase<ProfileSchedules> {

    private final SchedulesRepository schedulesRepository;

    @Inject
    public GetSchedules(SchedulesRepository schedulesRepository) {
        this.schedulesRepository = schedulesRepository;
    }

    @Override
    public Single<ProfileSchedules> buildUseCaseObservable() {
        return schedulesRepository.getSchedules();
    }

}
```

composer

- GetLinkDetails
- GetRetweetData
- GetUpdates
- GetUrlContent
- GetUrlDetails

content

- ChangeUpdatePosition
- DeleteUpdate
- DismissFeedItem
- GetFeedItems
- ShareUpdateNext
- ShareUpdateNow

profiles

- GetFriends
- GetProfiles
- GetSelectedProfile
- GetSelectedProfileId
- SaveSelectedProfile

schedules

- GetSchedules
- QueryTimezones
- UpdateSchedule
- UpdateTimezone

Interface Adapters

Activity

Interface Impl

Application Business Rules

Interface

Use Case

Interface Adapters

- Contains MVP architecture of our app

- Presenters contain ‘callbacks’

- Views define contract for implementation

- Mapper converts data from external form

```
public class UpdatesPresenter extends BasePresenter<UpdatesMvpView>
    implements SingleObserver<List<Update>> {

    @Override
    public void onSubscribe(Disposable d) {
        …
    }

    @Override
    public void onSuccess(List<Match> matches) {
      …
    }

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

}
```

Frameworks &
Drivers

Activity

Interface Impl

Interface
Adapters

Interface

Presenter

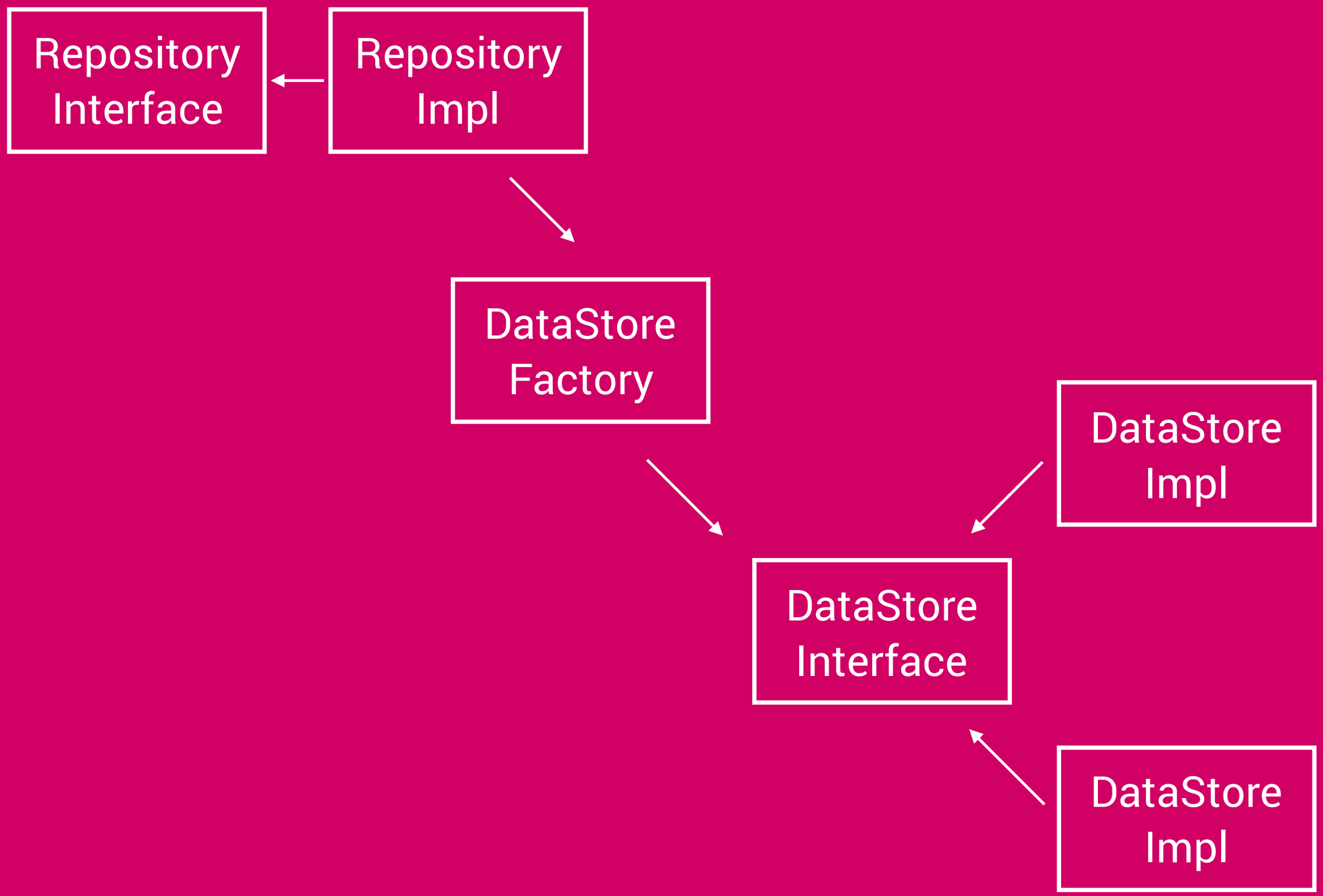
Frameworks & Drivers

- UI components (Activities, Fragments)

- Networking actions (Retrofit)

- Storage (Cache & DB)

- Repository Pattern for interacting with these


Repository Interface
Repository Impl
DataStore Factory
DataStore Impl
DataStore Interface
DataStore Impl

Enterprise Business Rules

Update

Profile

Feed

User

Schedule

Timezone

Application Business Rules

PostUpdate

UpdateEntity

EntityMapper

GetProfiles

FeedEntity

Interface

EditUpdate

Schedule Entity

Interactor Interface

User Interface

Network

Storage

Models

Mapper

Repository Interface

Presenters

Models

Reposit ories

Views

Mapper

Interface

Frameworks & Drivers

Interface Adapters

Layer Boundaries

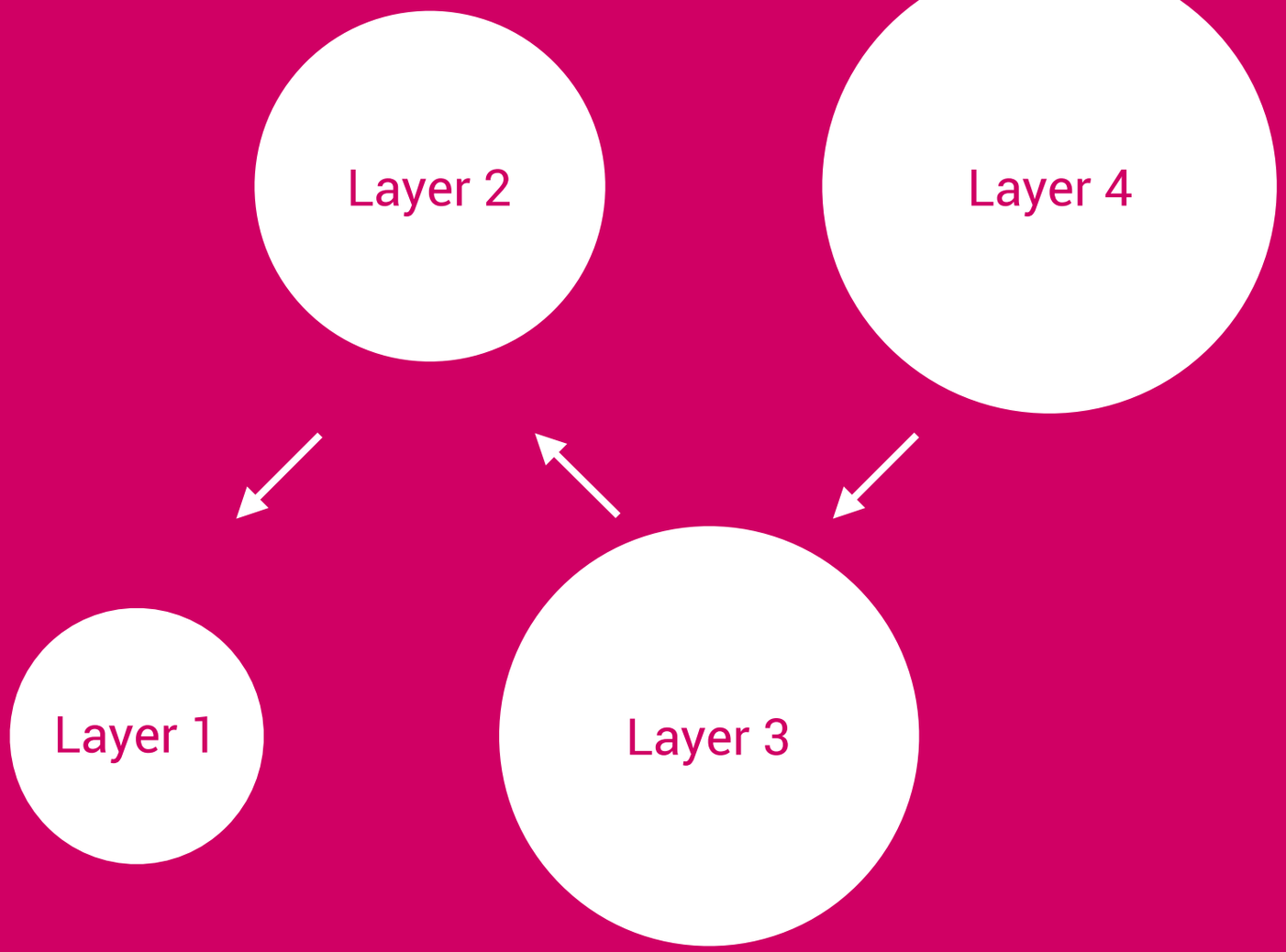
- Communication via abstractions (interfaces)

- No direct name references in inner layers

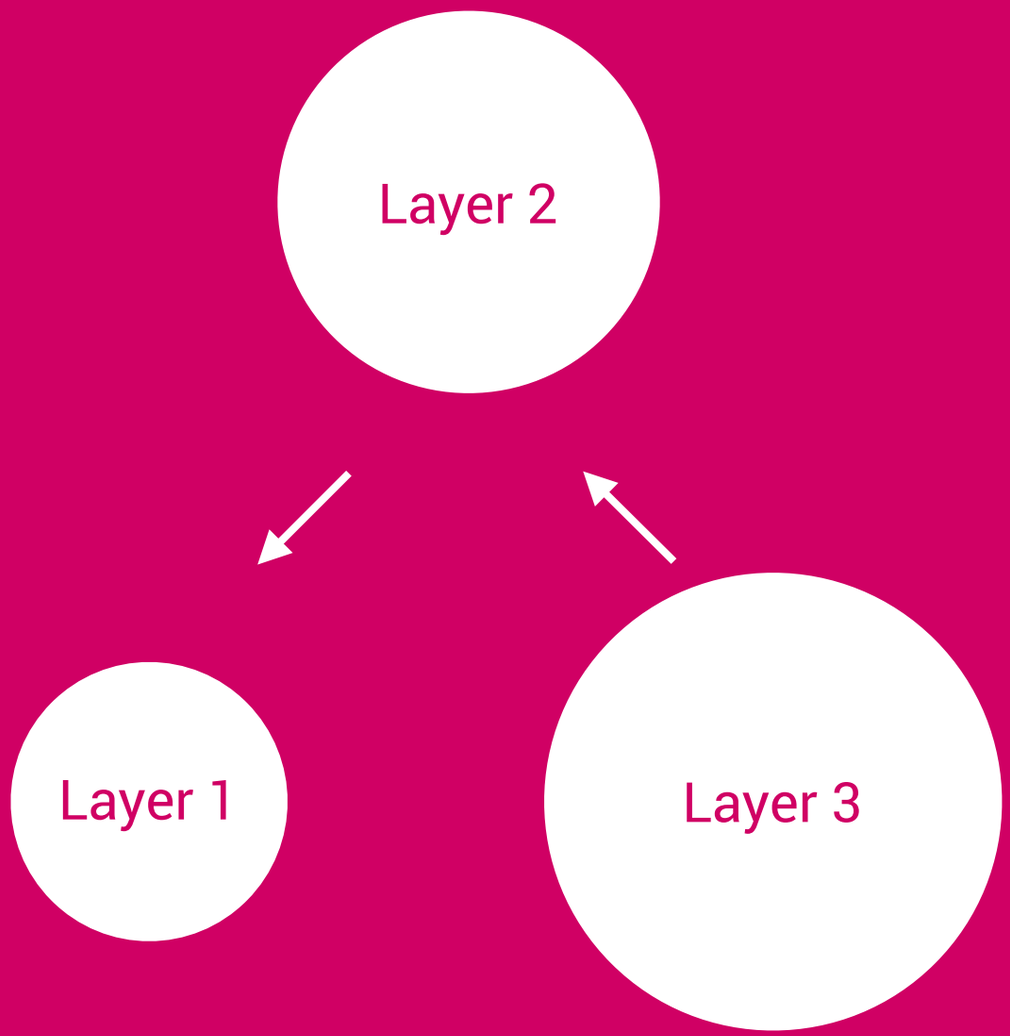
- Instead, inner layer provides this interface

- aka, The Dependancy Inversion Principle

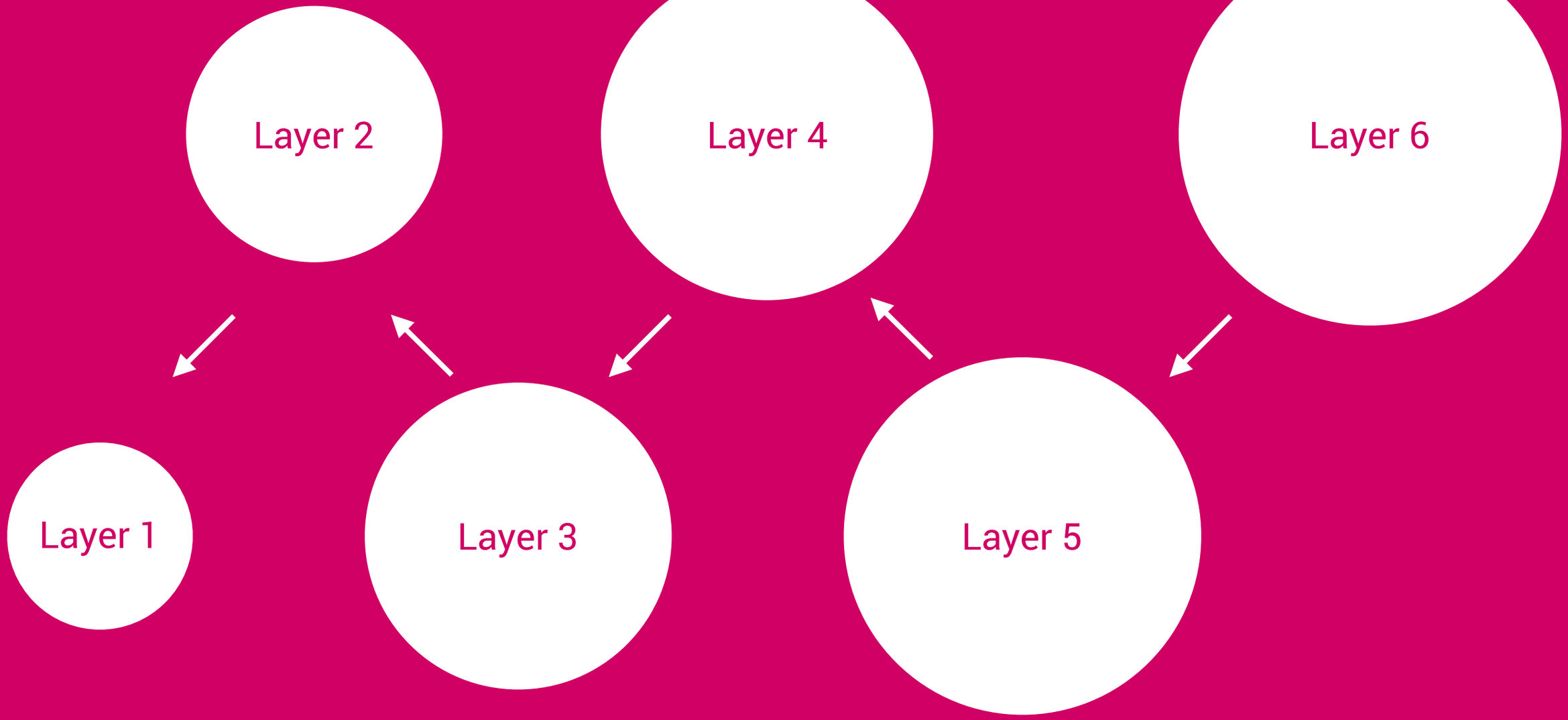
Not limited to these layers



Not limited to these layers



Not limited to these layers



Test Coverage

Enterprise Business Rules

Test model instantiation

Application Business Rules

Test use case builder methods

Test use case parameters

Test User Interface

Test Mapper class

Test View Models

Test Repositories

Test Preferences, DB etc

Frameworks & Drivers

Test Presenter classes

Test Mapper class

Test Models

Test Repositories

Test Data Store classes

Interface Adapters

Learnings

Advantages.

- High percentage of code coverage

- Incredibly easy to navigate package structure

- Easier to maintain

- Allows us to move faster

- Focused classes and test classes

- Seperation of concerns

- Define, Test, Stabilise before UI

- Futureproof implementations

- Clear discipline, good practice

- and more…

Disadvantages

- Adds initial overhead

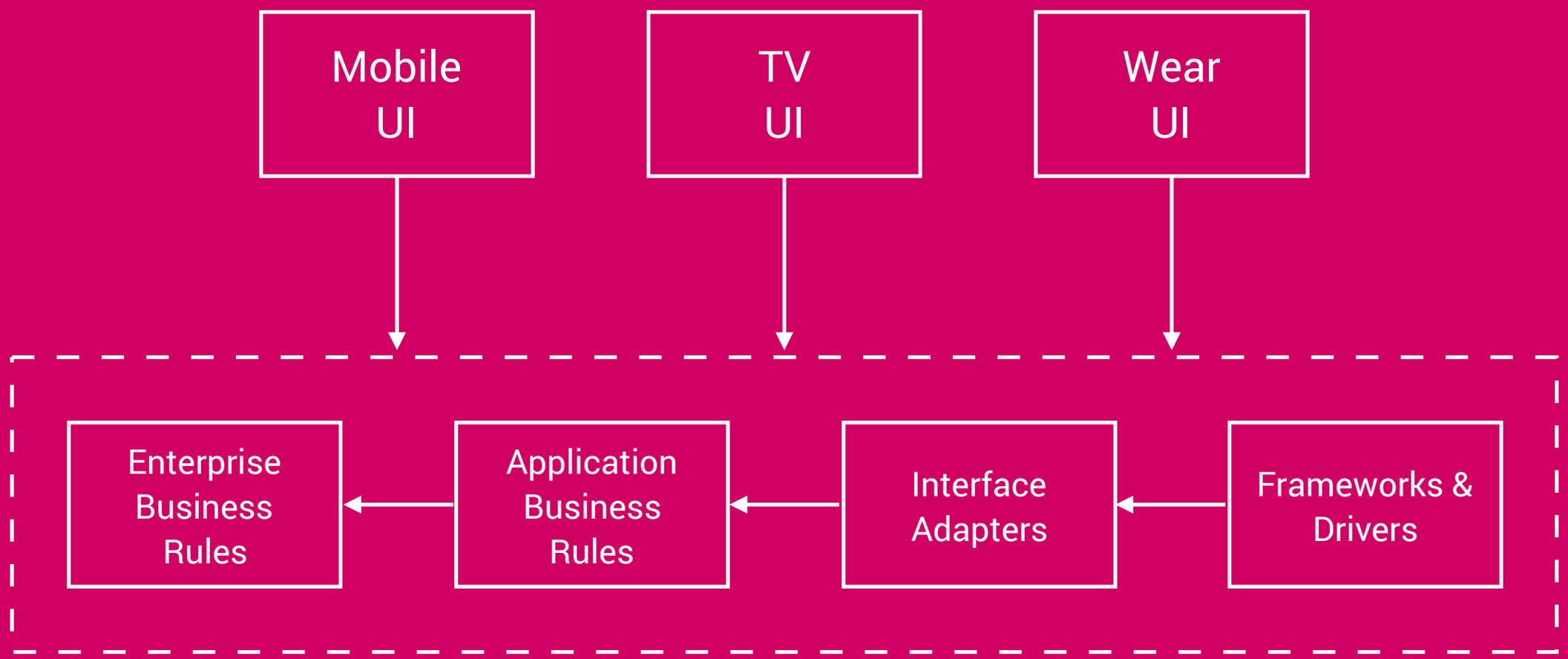
- Takes time to get used to

- Can feel overkill for some tasks

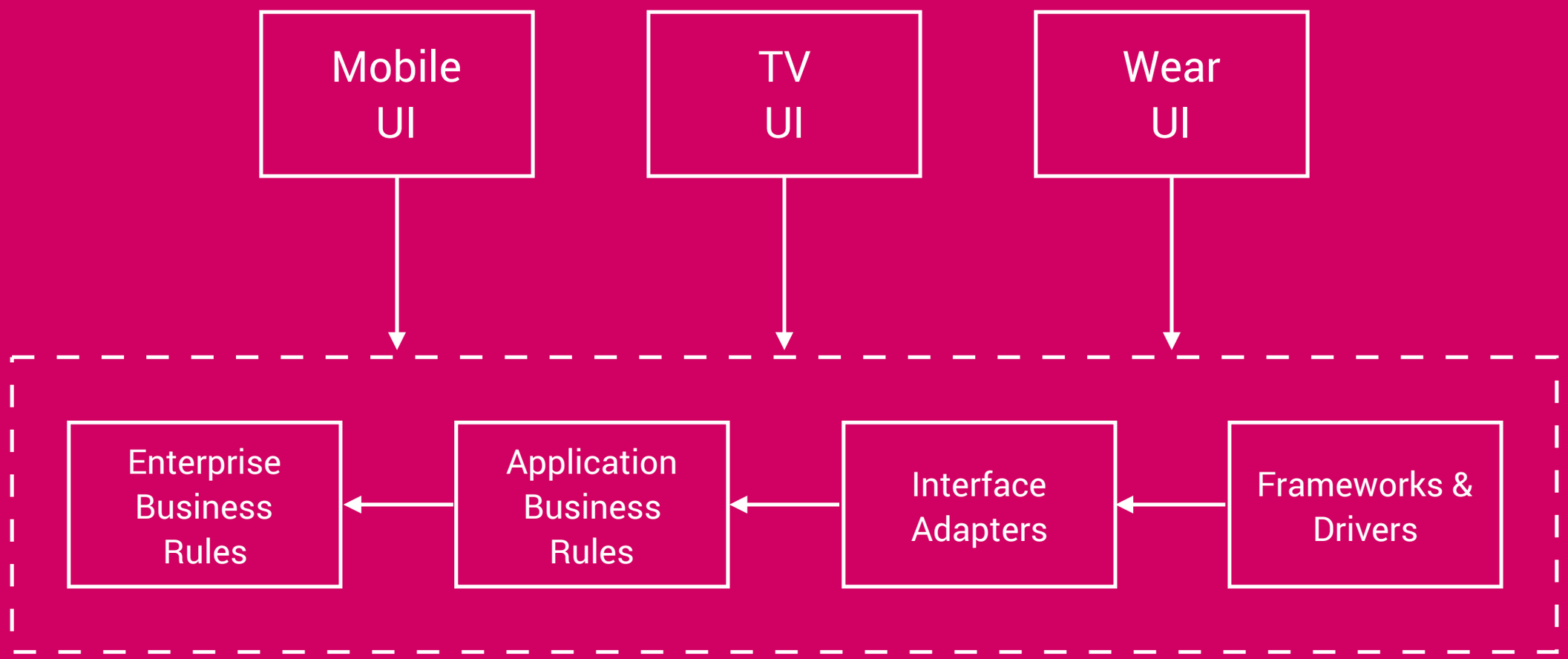
- Difficult to make decisions

- Conversion of models between layers

Winston


Mobile UI
TV UI
Wear UI
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers

Winston



@hitherejoe

hitherejoe

joebirch.co

Resources

https://overflow.buffer.com/2016/12/22/rebuild-android-composer/

https://github.com/googlesamples/android-architecture

Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin)