Building Apps with Android Architecture Components

THE IMPORTANCE OF SOFTWARE ARCHITECTURE PLANNING

Omri Erez

SOFTWARE ENGINEER

@innovationMaze | https://www.linkedin.com/in/omrierez/

Why is software architecture so important?

What characteristics does our software need to embrace?

Characteristics

- Maintainable & extendable
- Testable
- Understandable for new developers

Maintainable Software

The ability to fix a bug

Without introducing new bugs

The ability to fix a bug

Without it re-occurring in the future

The ability to fix a bug

With editing a low number of components

Extendable Software

The ability to add a new feature

With a minimum change of current components

The ability to add a new feature

Without changing the shape of the original architecture

Testable Software

The ability to test each component separately

Low maintenance effort for tests code

Efficiency in terms of testing effort and code coverage

Understandable for New Stakeholders

Low barrier of entry

New stakeholders can understand the project structure quickly

Easy to explain

All developers can easily explain the structure of the software

“Truth can only be found in one place: the code.”

Robert C. Martin

Evaluating the Complexity of Android Applications

Android Applications

- Can become very complex
- High number of core components
- High number of libraries

Android Applications

- Have restrictions from the OS
- Components have to comply

A Typical Android App

Activities	10
Data Adapters	5
Services	2
Fragments	4
Interfaces	10
Helpers / Utils	4
Content Provider	1
Database	1

A Typical Android App

Dependency Injection	API	Custom Views
Widget	Models	Custom Transitions

Interoperability Between Components

- Highly decoupled form each other
- Able to consistently communicate with others

Code Jedis & Trolls

Code Jedis

DANGER
Do not feed the troll

How can I find out if the code was written by a troll or by a jedi?

Indications for Jedis

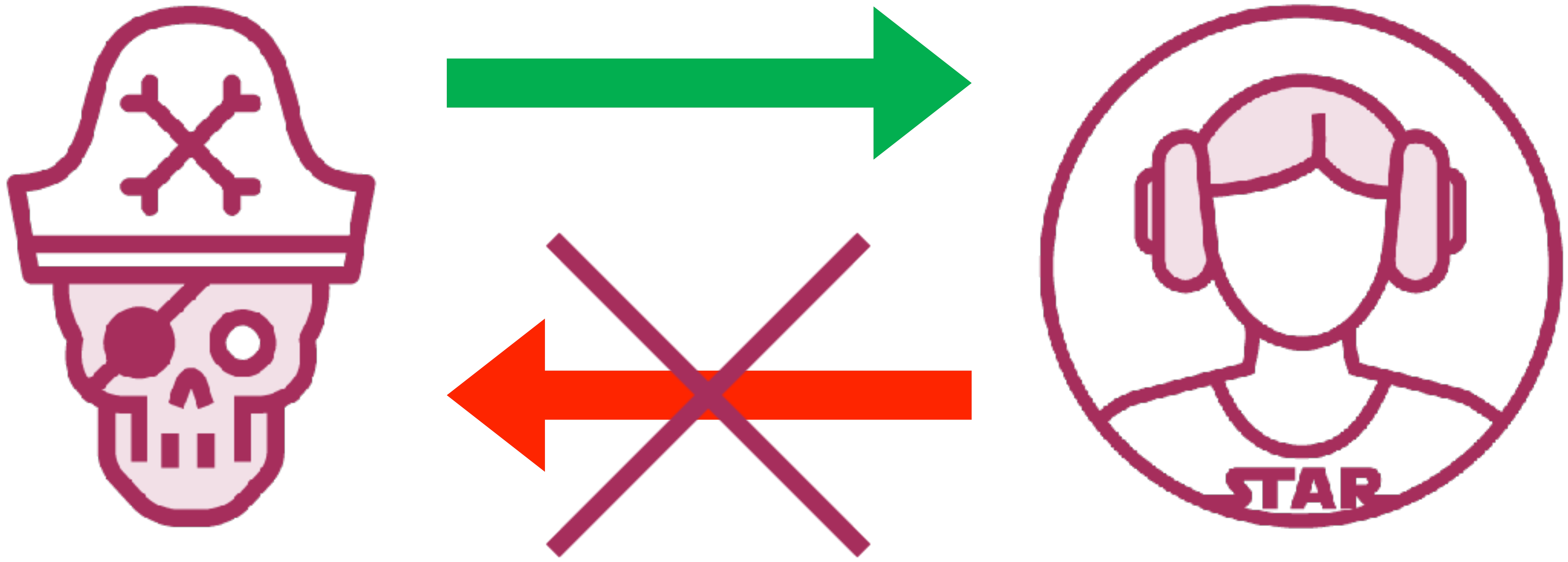
- Easy readable code
- Clear naming for classes & variables
- Following the SOLID principles for OOD
- Small classes

Indications for Trolls

- Not readable code = high number of comments

- Confusing structure and naming

- Very long classes


STAR

Goals

- Well defined structure & layers
- Well defined components
- Our code
  - Extendable
  - Maintainable
  - Testable

Remember

Good architecture saves valuable development time and as we all know, time = money

The SOLID Principles

- Firstly introduced by Robert C. Martin known as Uncle Bob
- First five object oriented design principles
- **Make it easy to write maintainable, extendable and testable code**

The SOLID Principles for Object Oriented Design

Single responsibility principle

Single Responsibility Principle

- Every class should have only one responsibility

- Responsibility = reason to change the class

- **Results in short components / classes**

Light

Magnifier

Hammer

Calculator

The SOLID Principles for Object Oriented Design

Single responsibility principle

Open / close principle

Open / Close Principle

- Should be open for extension
- Close for modification
- **Add new features using inheritance but shouldn’t change the existing class**

Open Close Violation

```java
public class ArrayProcessor {

    public void process(int [][] input)
    {
        for (int i = 0; i <input.length; i++) {
            switch(input[0][i])
            {
                case 0:
                    //do something 0
                case 1:
                    //do something 1
            }
        }
    }
}
```

Open Close

```java
public interface DigitProcessor
    {
        void process(int[] ints);
    }
public class ArrayProcessor {
    HashMap<Integer,DigitProcessor> mProcessors=new HashMap<>();
    public void addProcessor(int digit,DigitProcessor processor)
    {
        mProcessors.put(digit,processor);
    }
    public void process(int [][] input)
    {
        for (int i = 0; i <input.length; i++)
            mProcessors.get(input[0][i]).process(input[i]);
    }
   }
```

The SOLID Principles for Object Oriented Design

Single responsibility principle

Open / close principle

Liskov substitution principle

Liskov Substitution Principle

- A method that takes class Y as parameter
- **Must be able to work with any subclass of Y**

Liskov Substitution Violation

```
public interface Phone {

    void dial(int number);

}
```

```
public class AnalogPhone implements Phone {
    @Override
    public void dial(int number) {//some
logic}
}
```

```
public class SmartPhone implements Phone {
    @Override
    public void dial(int number) {
        if(isLocked())
            return;
        //some logic
    }
    public boolean isLocked()
    { //check if phone is locked}
    public void unlock()
    {

    }
}
```

Liskov Substitution Violation

```
public class PhoneManager {

    public void dial(Phone phone)
    {
        phone.dial(323485746);
    }
}
```

Liskov

```
public class PhoneManager {

    public void dial(Phone phone)
    {
        if (phone instanceof SmartPhone)
        {
            final SmartPhone smart=(SmartPhone) phone;
            if(smart.isLocked())
                smart.unlock();
        }
        phone.dial(323485746);
    }
}
```

Liskov

```java
public class SmartPhone implements Phone {
    @Override
    public void dial(int number) {
        if(isLocked())
            unlock();
        //some logic
    }

    public boolean isLocked()
    {
        //check if phone is locked
        return true;
    }

    public void unlock()
    {

    }
}
```

```java
public class PhoneManager {


    public void dial(Phone phone)
    {

        phone.dial(323485746);
    }
}
```

The SOLID Principles for Object Oriented Design

Single responsibility principle

Open / close principle

Liskov substitution principle

Interface segregation principle

Interface Segregation Principle

- Complex interfaces should be split

- **Complex interfaces makes it harder to extend smaller parts of our system**

Interface Segregation Violation

```java
public interface MultiPhone {
    void dial(int number);

    void calculatePlus(int a, int b);

    void calculateDivide(int a, int b);

    void calculateMultiple(int a, int b);

    void calculateMinus(int a, int b);

    void lightOn();

    void lightOff();
}
```

Interface Segregation

```java
public interface Phone {

    void dial(int number);

}
```

```java
public interface Flashlight {

    void lightOn();
    void lightOff();

}
```

```java
public interface Calculator {

    void calculatePlus(int a, int b);
    void calculateDivide(int a, int b);
    void calculateMultiple(int a, int b);
    void calculateMinus(int a, int b);

}
```

The SOLID Principles for Object Oriented Design

Single responsibility principle

Open / close principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Dependency Inversion Principle

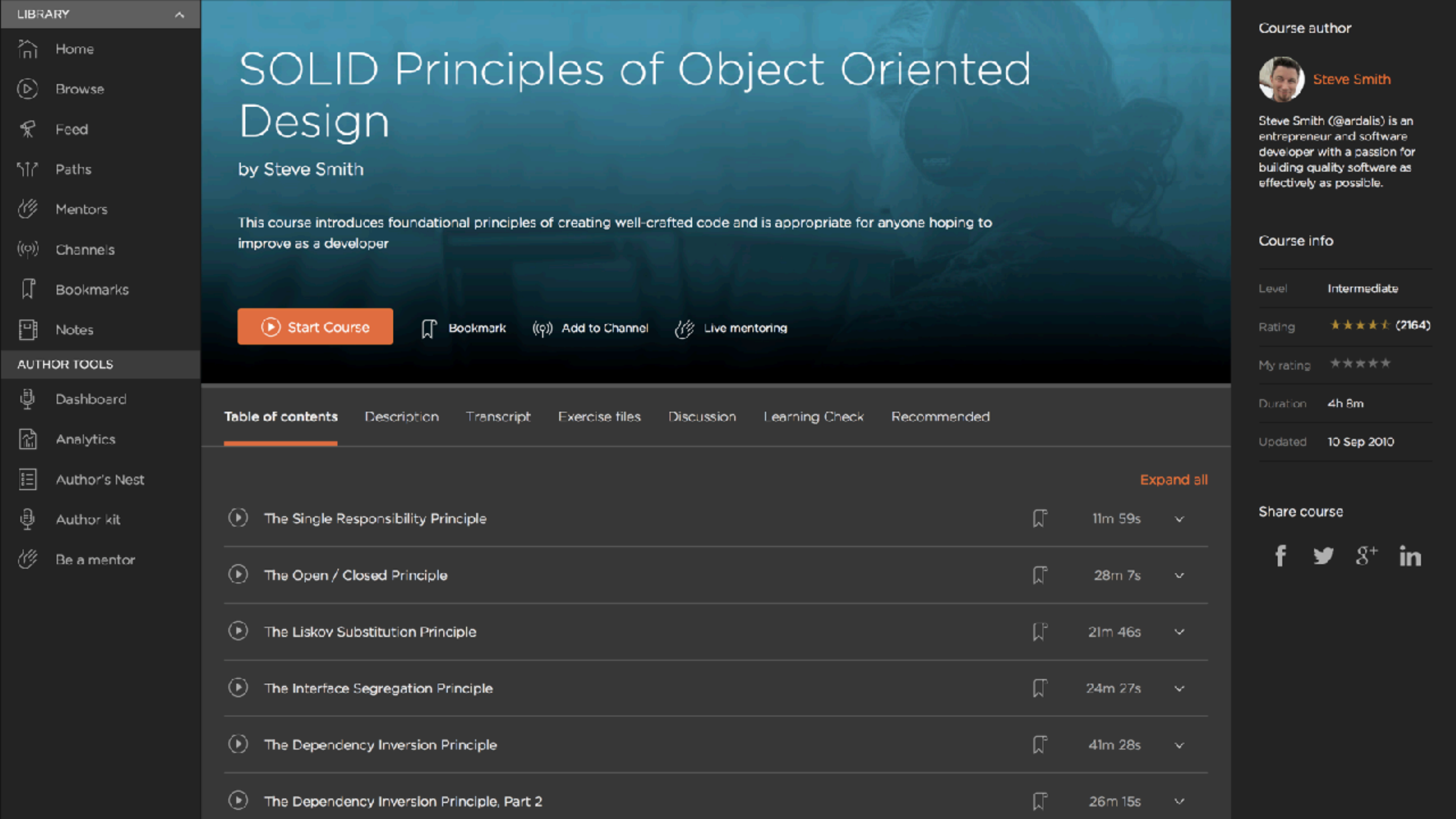
- No hidden dependencies
- Let the calling class create the dependency
- **Instead of letting the class itself create the dependency**

Dependency Inversion Violation

```
public class Bank {

    private Management mManage;
    private ClientsManager mClients;
    private AccountsManager mAccounts;

    public Bank() {
        mManage =new Management();
        mClients =new ClientsManager();
        mAccounts=new AccountsManager();
    }
}
```

Dependency Inversion

```
public class Bank {

    private Management mManage;
    private ClientsManager mClients;
    private AccountsManager mAccounts;


      public Bank(Management manage, ClientsManager clients,
                  AccountsManager accounts) {
        this.mManage = manage;
        this.mClients = clients;
        this.mAccounts = accounts;
    }
}
```



LIBRARY

Home

Browse

Feed

Paths

Mentors

Channels

Bookmarks

Notes

AUTHOR TOOLS

Dashboard

Analytics

Author's Nest

Author kit

Be a mentor

SOLID Principles of Object Oriented Design

by Steve Smith

This course introduces foundational principles of creating well-crafted code and is appropriate for anyone hoping to improve as a developer

Start Course | Bookmark | Add to Channel | Live mentoring

Table of contents | Description | Transcript | Exercise files | Discussion | Learning Check | Recommended

Expand all

The Single Responsibility Principle	11m 59s
The Open / Closed Principle	28m 7s
The Liskov Substitution Principle	21m 46s
The Interface Segregation Principle	24m 27s
The Dependency Inversion Principle	41m 28s
The Dependency Inversion Principle, Part 2	26m 15s

Course author

Steve Smith

Steve Smith (@ardalis) is an entrepreneur and software developer with a passion for building quality software as effectively as possible.

Course info

Level	Intermediate
Rating	★★★★★ (2164)
My rating	★★★★★
Duration	4h 8m
Updated	10 Sep 2010

Share course

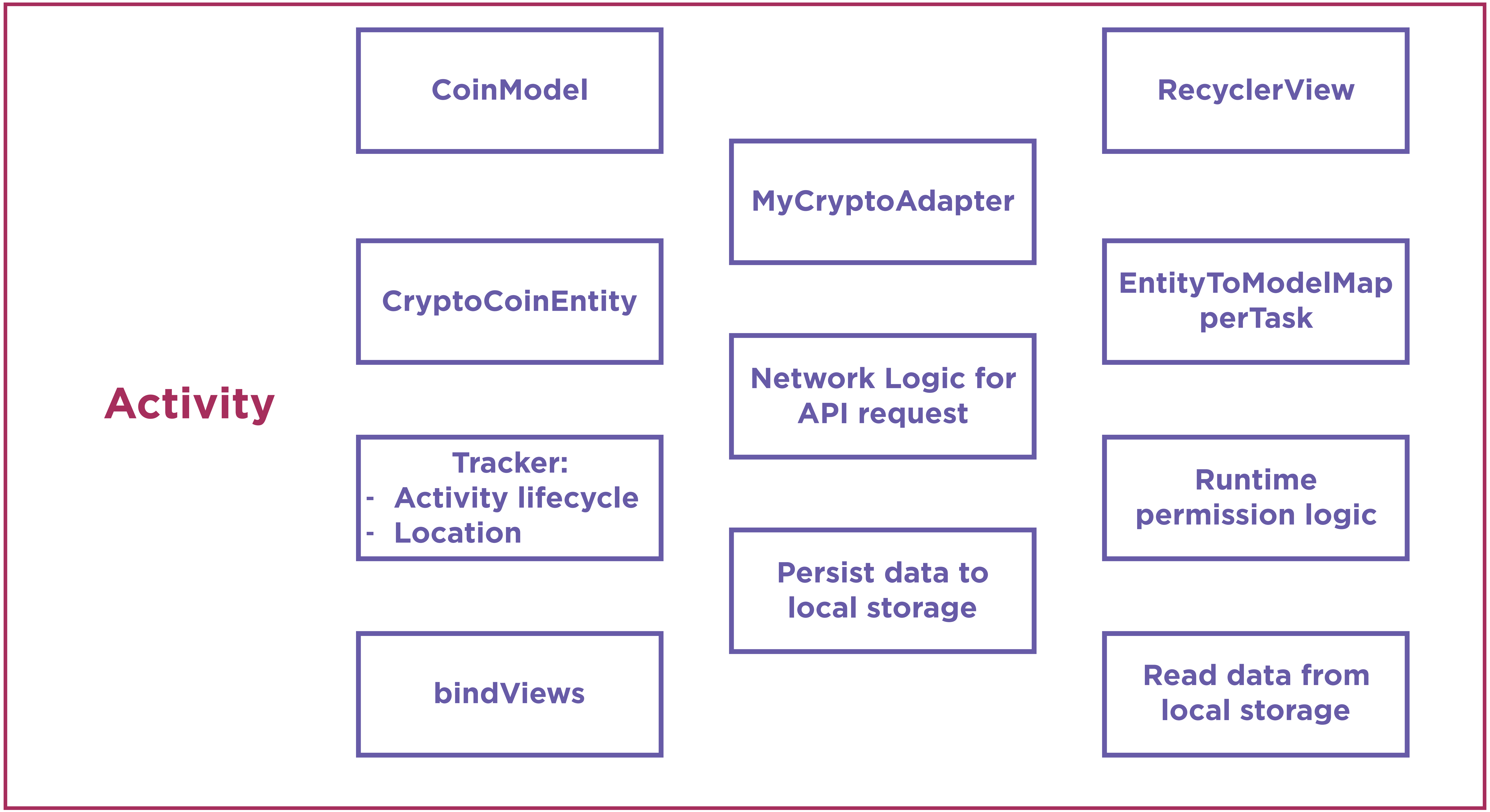
Demo

Crypto Boom App

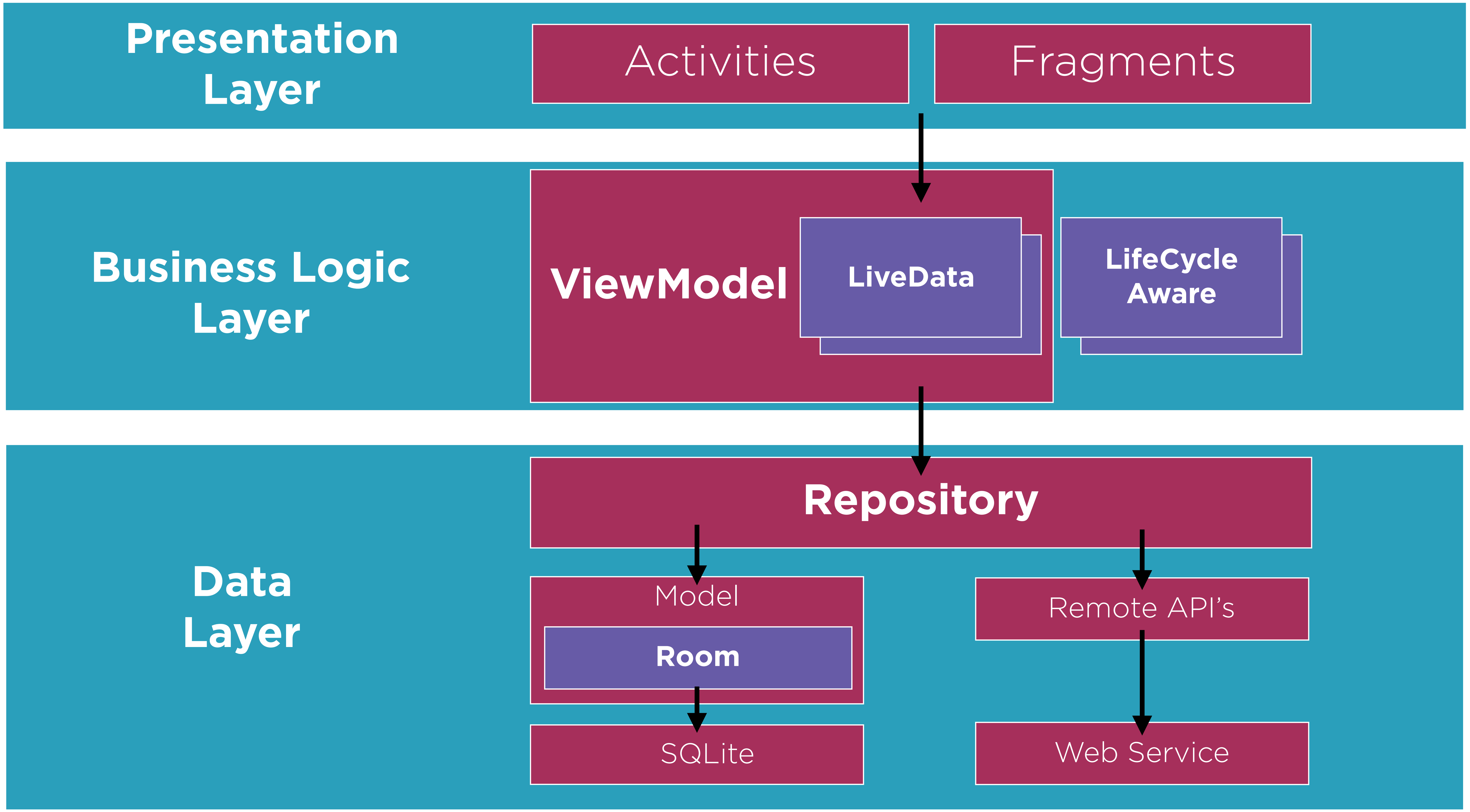
- Contains a list of crypto-currencies market data
- One God class
- Examine it’s initial state

DEMO SHOW THE APPLICATION IN IT’S INITIAL STATE

The Current “Architecture” of Our Demo App


Activity
CoinModel
CryptoCoinEntity
Tracker:
- Activity lifecycle
- Location
bindViews
MyCryptoAdapter
Network Logic for API request
Persist data to local storage
RecyclerView
EntityToModelMapperTask
Runtime permission logic
Read data from local storage

The Final Architecture of Our Demo App


Presentation Layer
Activities
Fragments
Business Logic Layer
ViewModel
LiveData
LifeCycle Aware
Data Layer
Repository
Model
Room
SQLite
Remote API's
Web Service

“Long lived software ALWAYS has legacy code and without well structured architecture, the technical debt will always grow”

The Code Jedi

Summary

Why architecture is important

- We want to produce:
    - Maintainable
    - Extendable
    - Testable

The SOLID Principles for Object Oriented Design

Single responsibility principle

Open / close principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Summary

Crypto Boom

- Our starting point:

  Initial messy “architecture”

- Our final goal:

  Well structured architecture