Structuring Android Applications

Architectural design patterns for rapid development and bug-free end products

Promoted by **Mr. Walcarius Stijn**
Written by **Anthony Madhvani**

Table of Contents

Licence

Introduction

Architectural patterns can oftentimes be difficult to grasp. They are clearly defined in numerous books, yet the knowledge and experience required to effectively apply them (and in the right places) can take many years to attain. A deep technical understanding of the software stack is required and at the same time the programmer must be able to visualize the code structure as a whole in order to predict and understand how it must be organized.

The gap between how easy it is to explain them and how difficult it is to apply them is not easily bridged.

This is especially true for new platforms such as Android. Yes, Android was first released seven years ago. Even though Android is not new in terms of how when it was first published, it could be considered very new indeed in terms of large codebases needing careful attention and maintenance -and most of all applications where rapid improvement and extension is the name of the game. Right now, innovation and a new sense of urgency is emerging about this in the Android development community.

Databinding, functional reactive programming, stores and dispatchers: an Android developer could be forgiven for not immediately knowing what any of those terms mean simply because they are so new to the community. If design patterns try to describe a way to organize code in the small, architectural patterns define code structure in the large. Their purpose is contrasted by the scope of their application. Structuring Android Applications is only concerned with the latter (yet some attention has been given to basic design patterns).

Technically, this thesis is about figuring out how to implement architectural design patterns on the Android framework. But it's real purpose is being able to match the stringent requirements put upon development teams and their applications with the right set of tools and abstractions.

Thesis Outline

This thesis is divided into two major parts.

The first part is preparatory and describes several design patterns and how they are applied in Android. This part also dedicates a chapter to the importance of correct tool and library usage.

- Chapter 1 (*Basic Design Pattern Knowledge*) describes several common design patterns that are involved in creating architectural patterns and application architecture in general.

- Chapter 2 (*Overview of Several Common Patterns*) gives the reader an introduction to architectural patterns and how they are applied in Android.

- Chapter 3 (*Tools, Testing and Libraries*) is a brief introduction into tools, testing and library usage.

- Chapter 4 (*The Example App*) lays out the reasoning behind the requirements of the example application and how quality measurement will be done.

Part two concerns itself with the implementation of the example application.

- Chapter 5 (*Building the Example App*) documents how the application was made and which considerations and problems arose while developing it.

- Chapter 6 (*Practical Analysis of Design Patterns*) will compare the various architectures that were implemented.

Glossary

Term	Definition
Design Pattern	A structured way of solving a common problem in software
Behavioral Pattern	A design patterns that decides how an application is built
Architectural Pattern	An overarching design pattern that decides how an application is architected
Dagger	A dependency injection library
RxJava	A library for developing in a functional reactive way
Square	A software development company

Basic Design Pattern Knowledge

Building Blocks

Even though non-architectural design patterns are not the focus here, it is still nessecary to know a couple of basic patterns in order to be able to implement any architecture. They will be briefly explained here in a practical way.

The Observer Pattern

The observer pattern is at the basis of many common architectures. In fact, it is so common that many languages have it included in their standard library (such as Java's Obverser class (Nystrom, n.d., chap. II.6)).

It can be thought of as a way to connect objects that want to be informed about the state of another object in a modular way. It will usually consist of an `observer` and a `subject`. It has a one to many relationship, since a given `subject` can have many `observers`.

The `subject` will have a method available to notify its observers and the observers will have a method available to subscribe to the events that the `subject` publishes. ("ACCU," n.d.)

```
ISubject
 void notifyObservers()
 void registerObserver(IObserver)


IObserver
 void notify(ISubject)
```

Say for example, an object called `Weather` exists that notifies anyone listening about changes in temperature. A `Weatherman` object would then subscribe to `Weather` in order to react to changes in weather.

```
Weather extends ISubject

Weatherman extends IObserver
  void notify(ISubject weather)
    makeWeatherAnnouncement(weather.getLatestWeather)

Weather weather = new Weather()
Weatherman weatherman = new Weatherman()

weather.registerObserver(weatherman)
weather.setLasterWeather("It's raining")
weather.notifyObservers()
```

Communication in this way allows for a high degree of decoupling while still providing certainty about being notified.

The Mediator Pattern

The mediator pattern is another pattern whose main benefit is increased decoupling of application

components. Using this pattern, a level of indirection is created such that objects do not communicate directly with each other.

```
interface IMediator
    void ChangeState()


 BrushDip extends IComponent
    void ChangeState(BrushType){
      ChangeBrush(brushType)
    }
```

A `mediator` will have one or more `components` that need to be informed about changes to any other `component` whenever one `component` sends a message. Where the observer pattern will have a *one to many* relation, the mediator pattern will have a *one to one to many* relation. Another big difference between the two is that by using the observer pattern, direct communication between objects still exists. Using a mediator, they become unlinked.

```
 Paint extends IComponent
    void ChangeState(PaintColor){
      paintOnCanvas(paintColor)
    }

 MakePaintingMediator extends IMediator
    void ChangeState(){
      brushDip.ChangeState(BrushType.random())
      paint.ChangeState(PaintColor.random())
    }

 //The brush no longer needs to directly communicate with the paint, increasing decouplin
g
 MakePaintingMediator artist = new MakePaintingMediator()
 artist.ChangeState()
```

Also note that the usefulness of the observer pattern can already be demonstrated here: using an observer, a mediator could notify all of its components about an event from any other component using a standard and easy to implement interface.

The Command Pattern

The command pattern encapsulates methods using an object in order to provide a standard way to handle events and data. Implementation wise a `Command` class will have a method `execute`.

```
interface ICommand
    void execute()
```

One of the biggest benefits is being able to queue a list of commands and letting another object execute them one by one. It would for example be possible to place a series of various network calls (loading images, loading HTML) in a list, and letting the object that receives those commands choose wether or not to request them one by one or in parallel.

```
class Command extends ICommand
  public Command(Sum sum, int one, int two)
```

```
void execute(){
  sum.add(one, two)
}
```

Since all commands are sent using the same type of class, it would be trivial to for example implement an `undo` method as well. Using this pattern, implementing a calculator that can easily do and undo operations becomes trivial.

```
class SumCommand extends ICommand
  public Command(int total, int one, int two)

  void execute(){
    previousValue = total
    total += one + two
  }

  void undo(){
    total = previousValue
  }

  SumCommand sumCommand = new SumCommand(5, 2, 2);
  Calculator.addCommand(sumCommand)

  //total = 5
  Calculator.executeLatestOperation()
  //total = 9
  Calculator.getLastOperations().undo()
  //total = 5
```
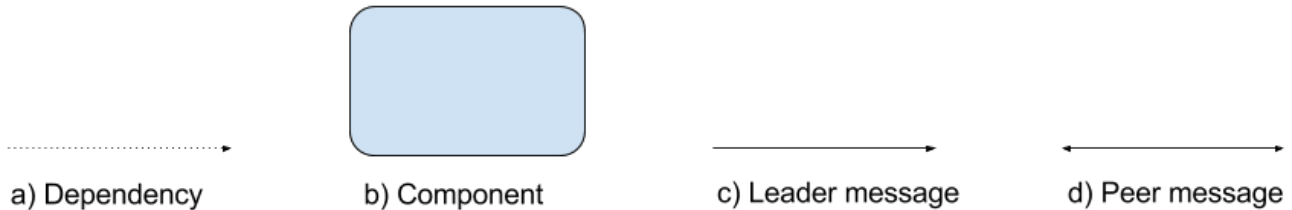
Overview of Several Common Patterns

This chapter will take a deep dive into various common architectural patterns and a high level view of their implementation in Android.

Diagram conventions

Conventions

a. A dependency indicates that one component *depends* on the other to provide it with data.

b. A component is an object or a set of objects that form a coherent whole. It is considered a black box with a simple input/output mechanism.

c. A leader message happens when one component can request changes to another component.

d. A peer message happens when two components are dependent on each other. A *unidirectional* flow of state is possible under this relationship.

- Sometimes a light background color will be used on a component to visualize how it is not equally important to the other components.

- I*Classname* signifies an interface.

MVC

Model, view controller.

Theory

MVC is by far the most common architecture. It is used by nearly every web framework and was first developed in the 1970's for Smalltalk, an early object-oriented programming language. Even though MVC was originally developed for usage *in the small* (where every piece of a view would have a separate controller and model), it is currently used for controlling the structure of entire views.

The general philosophy behind MVC is that the model (data), view (displaying) and the controller (routing between data and view) should be separated because they concern themselves with different responsibilities.

However under MVC components are quite linked. An interaction with the controller will make the model, providing data, communicate with the view. Because of this a requirement change for the view would require changes to the model as well. This becomes problematic when several views have to use the same model. While MVC was a big improvement on previous attempts, the coupling caused by it is quite severe.

That is not to say MVC is a useless abstraction model. It lends itself quite well to the web's model of communication for example or smaller applications where such coupling is not considered an issue. ("Techniques for Fault Tolerance in Software," n.d.,)
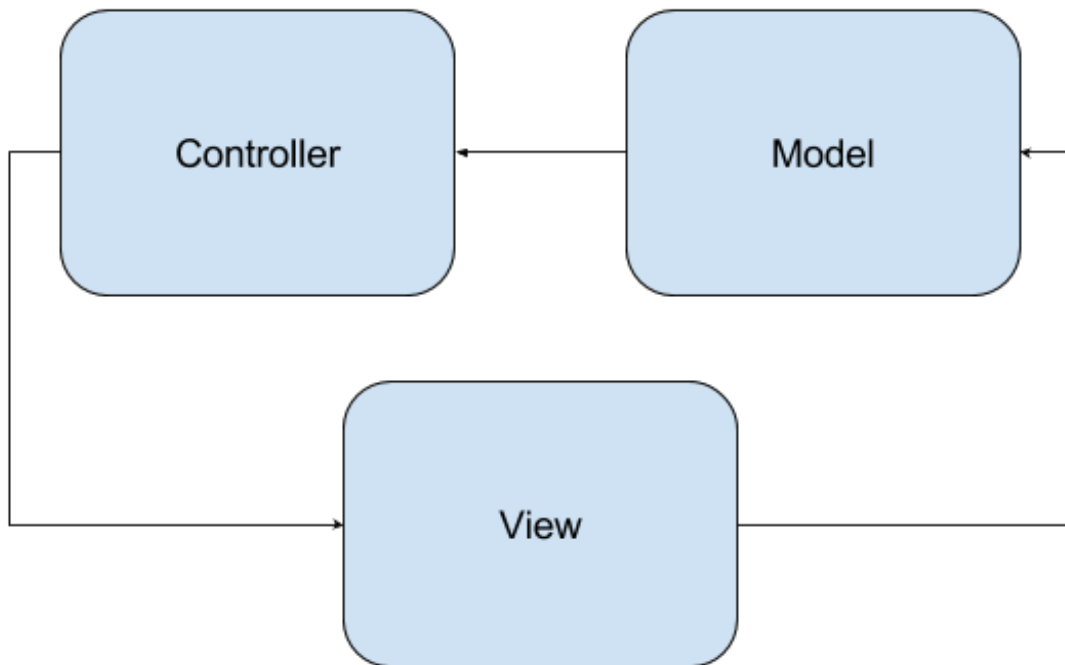
MVC

Implementation in Android

As was previously explained, the entry point in MVC is the controller and not the view. One some platforms like the web this is excellent, since all HTTP requests are handled by the server (the controller) before being displayed in the browser (the view). The mapping from theory to implementation is less straightforward in Android however: the entry point is an Intent which points to an activity (the view).

This makes it necessary to create workarounds or not follow along with the pattern too closely.
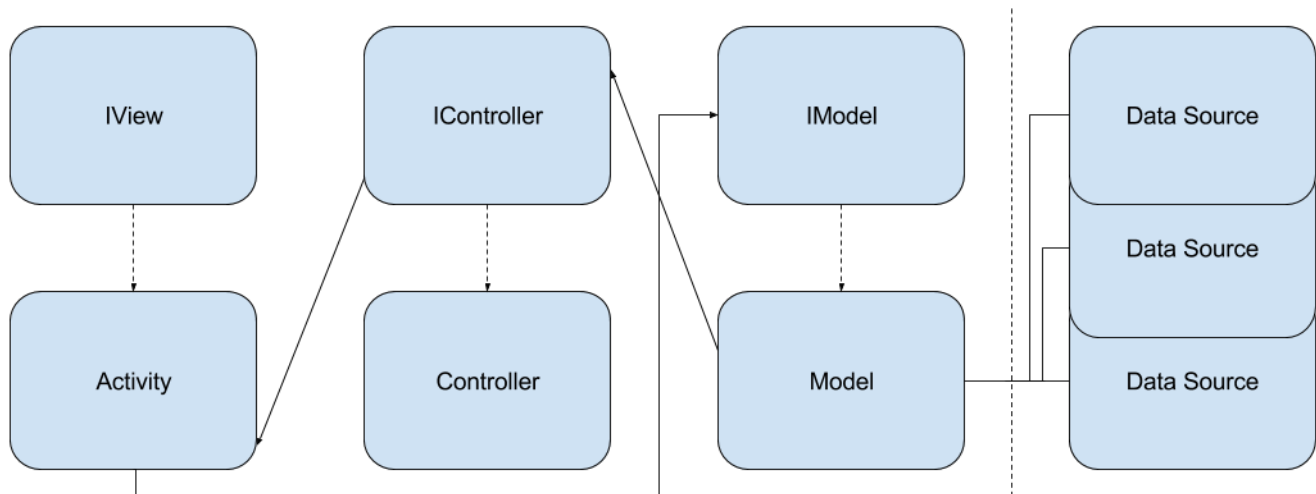
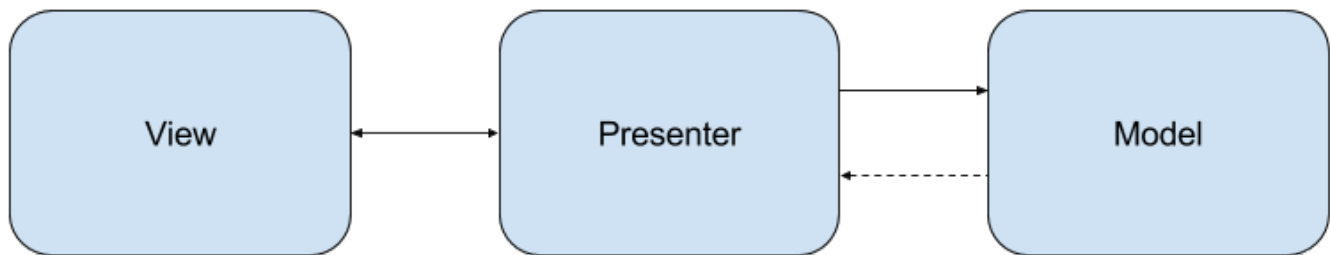MVC Implementation

MVP

Model, view, presenter.

Theory

Invented in the early 1990's, when software companies were seeing a huge increase in the complexity and required responsiveness of views. Besides increasingly complex interfaces, views had to adapt much faster to business requirements as well. The invention of MVP was one that arose out of a need to further

decouple a data source from the view even further.

MVP is essentially a variation on MVC using a different control flow. However, a big improvement on MVC is that under MVP the view has absolutely no knowledge of the model. This is what *presenter* takes care of: this component prepares data from the model for the view. While this seems like a small difference, the separation between view and model ensures that no dependencies can arise between them. This in itself is an important improvement that increases decoupling.

Unlike MVC, the entry point is the presenter. The view and the presenter have a one-on-one mapping which means they both share knowledge of each other. ("MVP for Android," 2014–2014-04-15T15:19:55+00:00)

MVP

Implementation in Android

Using MVP, it is possible to strictly obey the pattern guidelines since user interaction originates with the view. It should also be noted that because views and presenters have a one-on-one mapping, using interfaces is not immediately required.
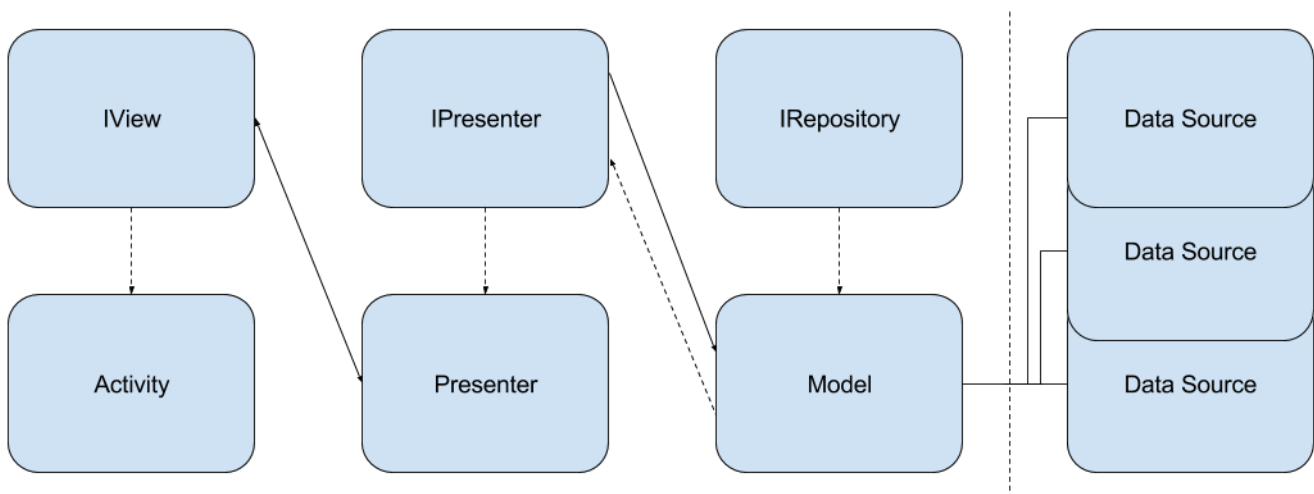
MVP Implementation

MVVM

Model, viewmodel, view.

Theory

MVVM was originally developed by Microsoft in order to benefit from WPF's event driven architecture. However, on the Android platform it is possible to work with events and *observers* as well. The viewmodel is often called a *value converter* because it prepares the data from the model for the view. By using an event-based system, the viewmodel can send changes to the view (which *observes* the viewmodel's properties). However, the viewmodel should have no knowledge of the view. This has a number of

important ramifications, most of all that *databinding* becomes a necessity.

Databinding synchronizes an observer with a subject by sending evented commands. These events ensure that the view and the viewmodel have exactly the same state.

When using MVVM, the entry point is the view.

Model-view-viewmodel is perhaps the most careful in terms of component communication. Unlike MVP, the view is kept as *dumb* as possible, only displaying the information that the viewmodel provides.

What this means is that unlike MVP, a viewmodel could potentially be shared with many views. This strict separation allows for a very decoupled architecture. Contrast this with a view-presenter relationship where the view is still responsible for manipulating information and state.
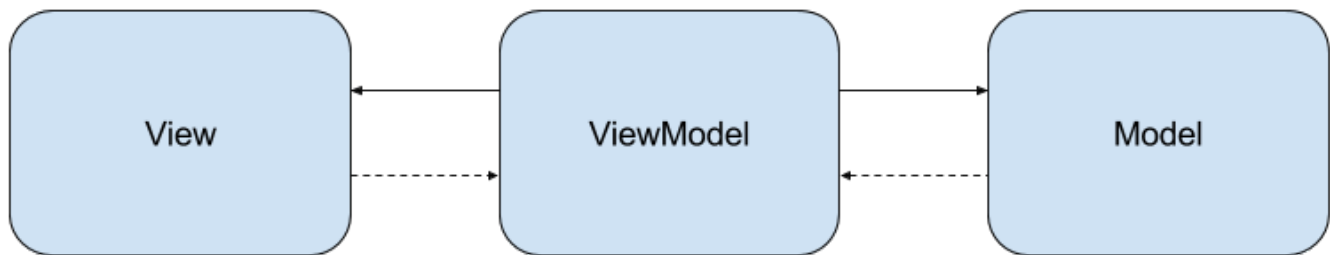
MVVM

Implementation in Android

Similar to MVP, the MVVM patterns fits quite well into Android. Of interest is also that a databinding library was recently introduced to Android which makes it much easier to use MVVM. Another important fact to note is that the Activity itself becomes nothing more than a connector to the layout resource file and the object which holds on to the Android lifecycle.
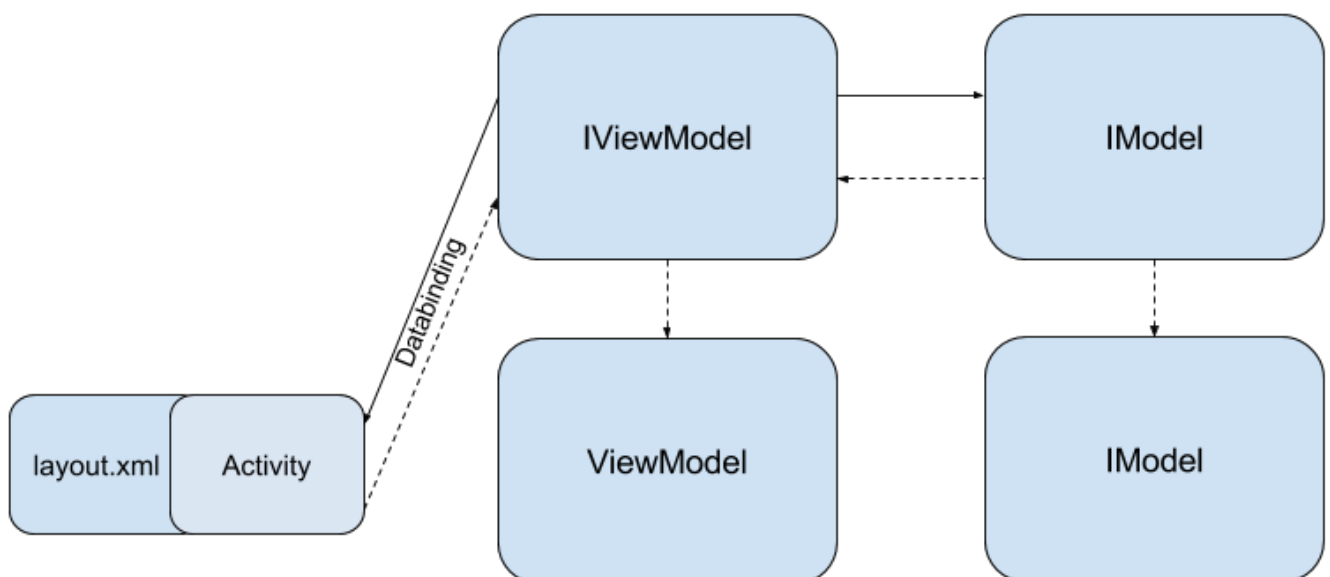
MVVM Implementation

The Example App

In order to give a fair representation of each development method's benefits and downsides, a single app will be built each time using a different design pattern. This will, among other things, make it possible to give accurate assessments in terms of performance and development speed.

When choosing what kind of application will fit that purpose, common application usage is the most important qualifier. Developing an application with a very uncommon or niche purpose would only be useful as a theoretical exercise.

What it should do

- Asynchronously load images

- Make multi-threaded network calls

- Consume an API

- Make adjustments to the system

- Use a service to give periodical updates to the user

- Implement and use a custom view

- Implicitly call other activities, both internal and external

With these requirements in mind, an example app has been chosen.

The app: WikiaArt Image Downloader

WikiArt is a website which serves as a central repository for art throughout the ages. Relevant to the app requirements, it has an API and contains high-resolution imagery. Using their API, an app will be constructed that allows users to browse art and choose a new wallpaper.

Flow

The following figure shows how the user will interact with the application and go from one activity to the next.
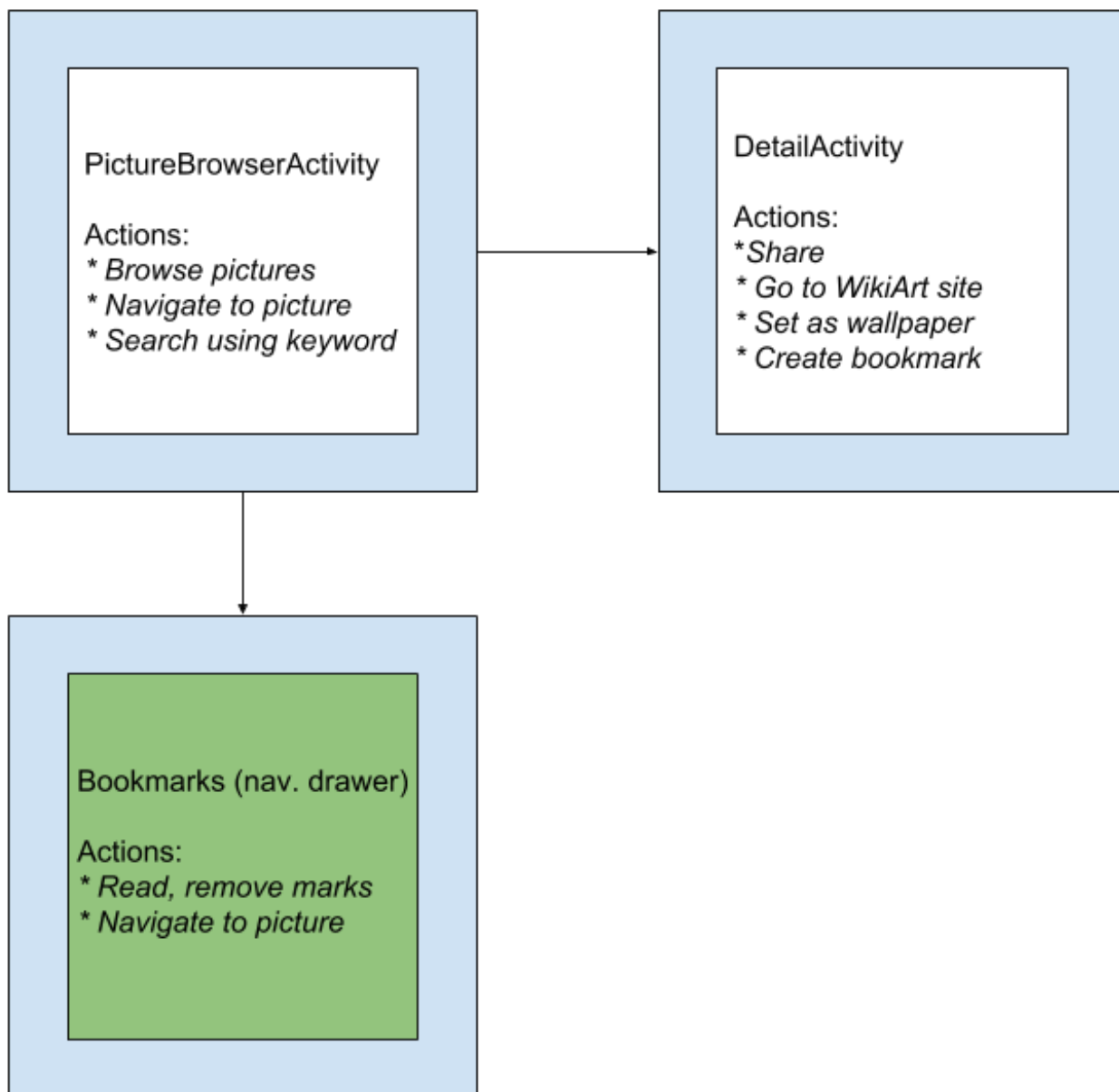
Application flow

Quality measurement

This list of metrics was adapted from *Code Quality: The Open Source Perspective,* which won the 2007 Software Development Productivity Award. (Beizer, 2003)

Efficiency and resource utilization

Efficiency will be measured using Android Studio's built-in profiler, an extremely useful but underused set of tools. This has a number of benefits including that every pattern can be tested on a standard set of devices. This allows us to ignore device-specific Android versions, such as Samsung, which tend to have some differences from the standard OS ("Icechen1/androidcompat," n.d.) ("There is a special place for Samsung in Android hell - Anas Ambri," n.d.).

In order to provide examples that are both usable in the real world and easy to demonstrate, only the latest version of Android will be tested on. Currently this is 6.0.

Decoupling

> Decoupling is the process of separating them so that their functionality will be more self contained.

Testability

> Testability is the degree of difficulty of testing a system. This is determined by both aspects of the system under test and its development approach.

Fault tolerance

> The assumption that the system has unavoidable and undetectable faults and aims to make provisions for the system to operate correctly even in the presence of faults.

Extensibility

> Extensibility is the capacity to extend or stretch the functionality of the development environment — to add something to it that didn't exist there before.

Verbosity of code

> Good code should be easy to comprehend at a glance. This is easier if most of the characters directly serve the purpose of the code.

Due to Java's age, a lot of "ceremony" is sometimes required to achieve what a modern language can do more easily. This is especially evident when using an older version of Java like Android does. While certain projects want to remove Java from the picture entirely, such as Kotlin, there is no evidence to support that Google will move away from Java anytime soon. So improving on the efficiency and speed of Android development using standard Java is a top priority. This becomes all the more important when deciding how a whole application should be structured.

Complexity of implementation

If a certain design pattern look promising but proves difficult to implement it simply might not be worth the extra effort, since a big motivator for developing in a structured way is keeping duplication to a minimum and increasing development speed.

A point could also be made that the amount of boilerplate a developer must write in order to create something usable has a direct correlation with the amount of bugs that creep up in a project.

Building the Example App

This chapter documents how the example application was developed using various architectures. Besides a working implementation, a number of experiences and conclusions are also made. Readers are encouraged to follow along and develop the sample application as well.

Online repository

The Google way

Project setup

Development

Conclusion

MVP using Mortar

Project setup

Development

Conclusion

MVVM

Project setup

Development

Conclusion

Flux

Project setup

Development

Conclusion

Practical Analysis of Design Patterns

Having developed the application, the various architectures will now be examined one by one according to a number of quality indicators. In conclusion, advice will be handed out for deciding which architecture would best fit a project.

Resource usage

MVP

Google way

MVVM

Flux

Android lifecycle management

MVP

Google way

MVVM

Flux

Testability

MVP

Google way

MVVM

Flux

Verbosity

MVP

Google way

MVVM

Flux

Comparison

Small app

Medium-size app

Large app

Conclusion

References

ACCU : The Philosophy of Extensible Software. (n.d.). Retrieved February 24, 2016, from http://accu.org/index.php/journals/391

Beizer, B. (2003). *Software Testing Techniques*. Dreamtech.

Icechen1/androidcompat. GitHub. (n.d.). Retrieved February 24, 2016, from https://github.com/icechen1/androidcompat

MVP for Android: How to organize the presentation layer. Antonio leiva. (2014–2014-04-15T15:19:55+00:00). Retrieved February 24, 2016, from http://antonioleiva.com/mvp-android/

Nystrom, B. (n.d.). Design patterns revisited · game programming patterns. Game programming patterns. Retrieved February 21, 2016, from http://gameprogrammingpatterns.com/design-patterns-revisited.html

Techniques for Fault Tolerance in Software. (n.d.). Retrieved February 24, 2016, from http://srel.ee.duke.edu/sw_ft/node5.html

There is a special place for Samsung in Android hell - Anas Ambri. (n.d.). Retrieved February 24, 2016, from http://verybadalloc.com/android/2015/12/19/special-place-for-samsung-in-android-hell/