Table of Contents

Licence

Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

This is a human-readable summary of (and not a substitute for) thelicense.

You are free to:

- Share — copy and redistribute the material in any medium or format
- Adapt — remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- NonCommercial — You may not use the material for commercial purposes.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Preface

My motivation behind this thesis was first and foremost the realization of my own shortcomings in developing software.

As I took on ever more challenging projects, the complexity of structuring software -and structuring it well- became clear. This problem was not as easily solved as I had hoped, because for every pristine and seemingly perfect design pattern a range of messy concrete details were hidden just out of sight.

Because simply moping about it turned out tiresome, I decided to translate this frustration into something others might find useful: a way to implement these abstract structures onto a physical device. Something tangible.

A word of thanks goes out to my promotor Mr. Walcarius and the lovely developers of `reddit.com/r/androiddev` `androidchat.slack.com` who dutifully served as my collective "second reader". I'm not sure how this project would have turned out without the help of everyone who supported and guided me.

As this thesis was written with implementation in mind, I sincerely hope you will find it useful in your daily work (even if just in a small way).

Abstract

"Structuring Android Applications" entails specifically the usage of architectural design patterns for the purposes of creating decoupled, well-structured, stable and extendible software for the Android platform.

In short

An analysis is made of several architectural design patterns and how they are implemented on Android. Ultimately the conclusion is made that the recently developed "Flux" framework by Facebook is a suitable candidate for many common issues faced in developing for mobile devices. Despite its lack of community support it can readily be implemented using only a few basic libraries.

Besides this unlikely option, a point is made that MVVM can also be an excellent solution now that the Data Binding Library has had its first stable version released. Especially so for applications which share much common logic between views with only slightly differing layouts.

Lastly, after analysis the conclusion is made that MVP is best suited for larger teams who require very fine grained control of every specific part of an application.

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 tooling 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 is made and which considerations and problems arose while developing it.
- Chapter 6 (*Practical Analysis of Design Patterns*) will compare the various architectures that are implemented.

Glossary

Term	Definition
ADB	Android Developer Bridge
ADM	Android Device Monitor, provides additional tooling for Android
Android Studio	An IDE created for the development of Android applications
Annotations	Meta-information appended to code structures
Antipattern	Code smell, a code structure which may indicate unsound coding practices
Architectural Pattern	An overarching design pattern that decides how an application is architected
Asynchronous function	A function which does not block the execution of a program
Behavioral Pattern	A design patterns that decides how an application is built
Boilerplate	Initialization code deemed repetitive
Dagger	A dependency injection library
Dependency graph	A directed graph declaring how one objects possibly depends on other objects
Dependency injection	Declaratively requesting code which has a certain behavior instead of concrete classes and objects
Design Pattern	A structured way of solving a common problem in software
Functional programming	Computation as pure mathematical functions without side effects
Generated code	Code created without a programmer's direct involvement
Global state	State accessable by most every part of an application
GoF	Shorthand for "Design Patterns: Elements of Reusable Object-Oriented Software", a well known book on design patterns
Overhead	Performance-decreasing load which is not directly associated with the true function of a component
RxJava	A library for developing in a functional reactive way
Square	A software development company
Square	A software development company
Tumblr	A blogging website
Unit testing	Testing the functionality of a single component in isolation
View	A rectangular on-screen object which handles user interaction and layout

Figures, Tables and Code Samples

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.

Basic Design Pattern Knowledge

Building Blocks

Even though non-architectural design patterns are not the focus here, it is still nessecary to know at least these 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.(Gamma, Helm, Johnson, Vlissides, & Booch, 1994)

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.(Gamma et al., 1994)

```
 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, incre
asing decoupling
 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.(Gamma et al., 1994)

```
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.(Nystrom, n.d.)

```
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
```

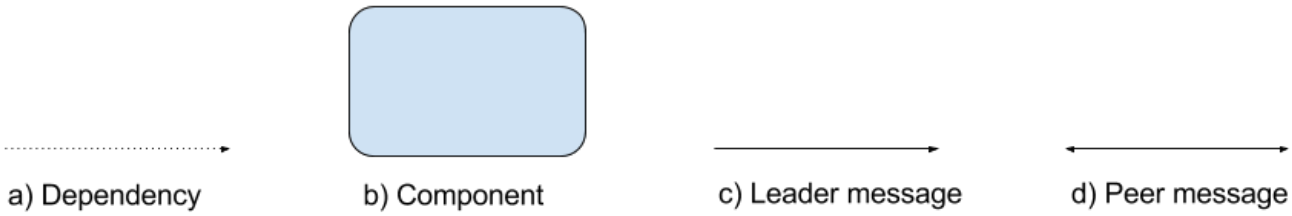
Overview of Several Common Patterns

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

Throughout this chapter, it is important to keep in mind that design patterns are not rigorously defined frameworks. This is in contrast to what most students are taught, leading to misunderstandings where for example MVC is considered nothing more than a concrete ASP technology.

Rather, architectural patterns are a way of *structuring and organizing code*. This becomes all the more obvious when trying to convert those conventions into functioning code.

Diagram conventions



(Fig, 14.1) 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.

- **Will not be implemented due to extremely low traction**

Theory

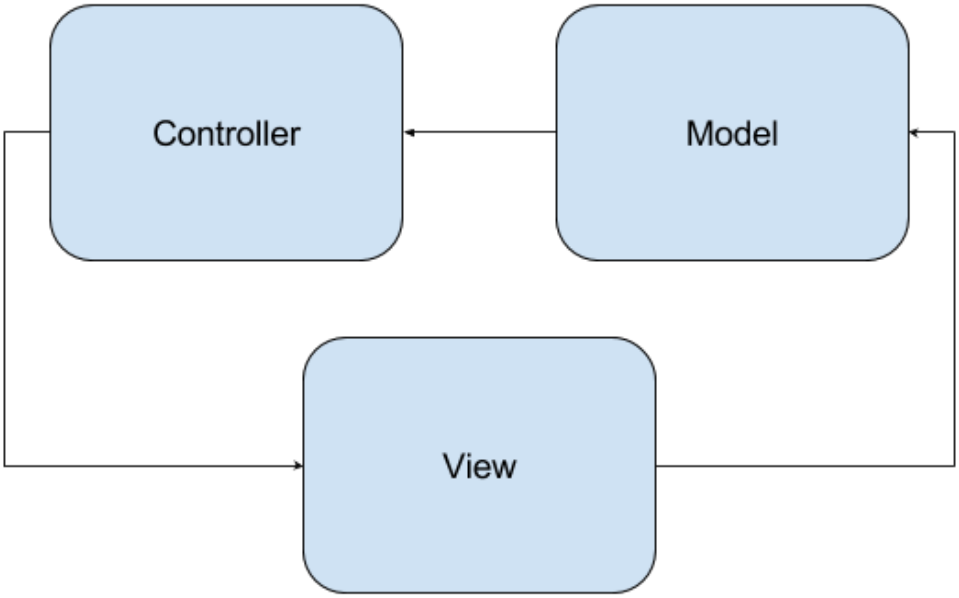
MVC is by far the most common architecture. It is used by nearly every web framework and was first developed in the 1970's for Smalltalk, an early object-oriented programming language. Even though MVC was originally developed for usage *in the small* (where every piece of a view would have a

separate controller and model), it is currently used for controlling the structure of entire views.(C. Martin, n.d.)

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.,)

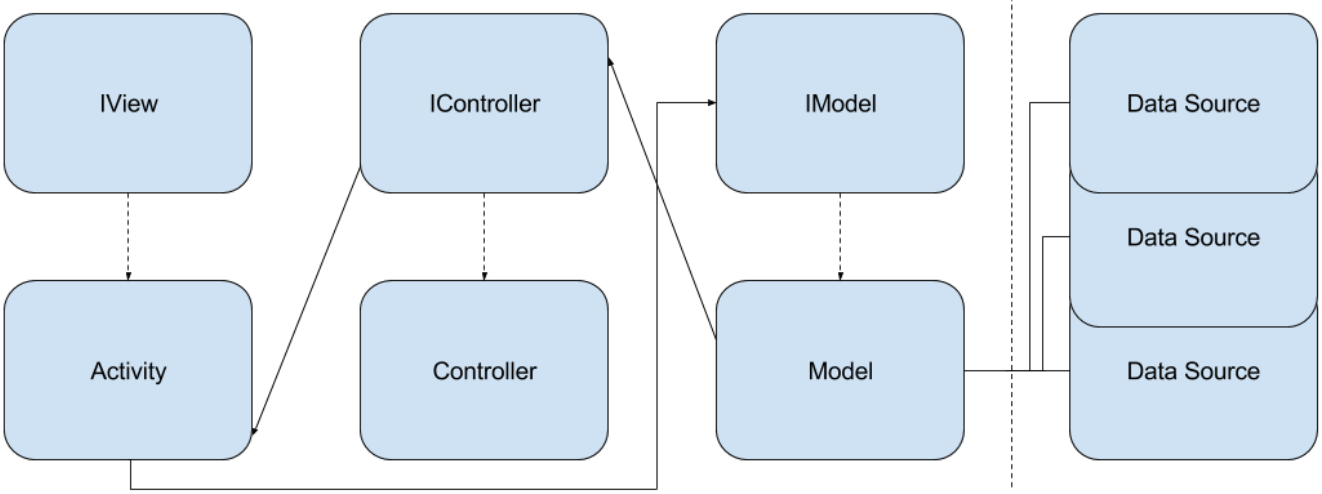


(Fig, 15.1) MVC

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 or use a very loose interpretation of what consitutes MVC.



(Fig, 16.1) MVC Implementation

MVP

Model, view, presenter.

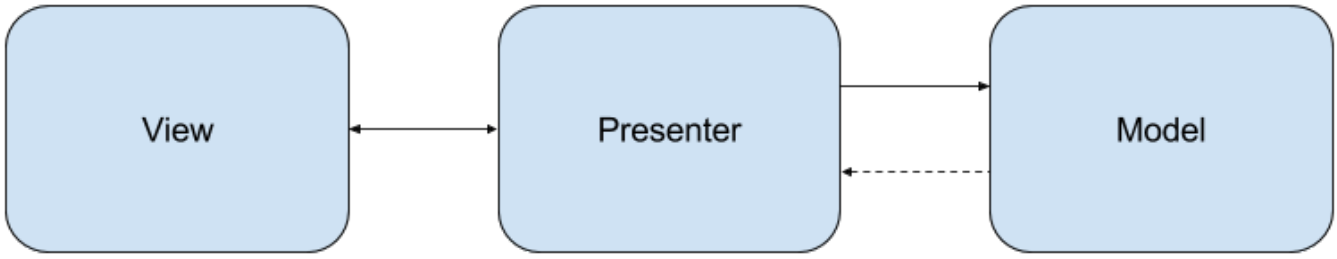
- **Will be implemented**

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.(Richards, n.d.)

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.

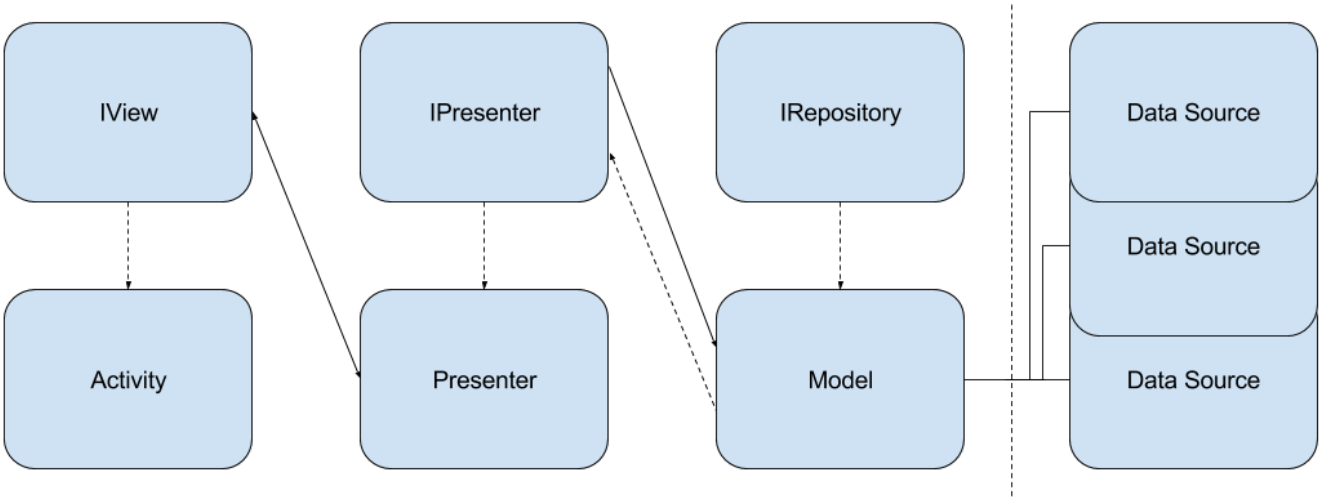
Unlike MVC, the entry point is the presenter. The view and the presenter have a one-on-one mapping which means they both share knowledge of each other. (“MVP for Android,” 2014–2014-04-15T15:19:55+00:00)



(Fig, 16.2) 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.



(Fig, 17.1) MVP Implementation

MVVM

Model, viewmodel, view.

- **Will be implemented**

Theory

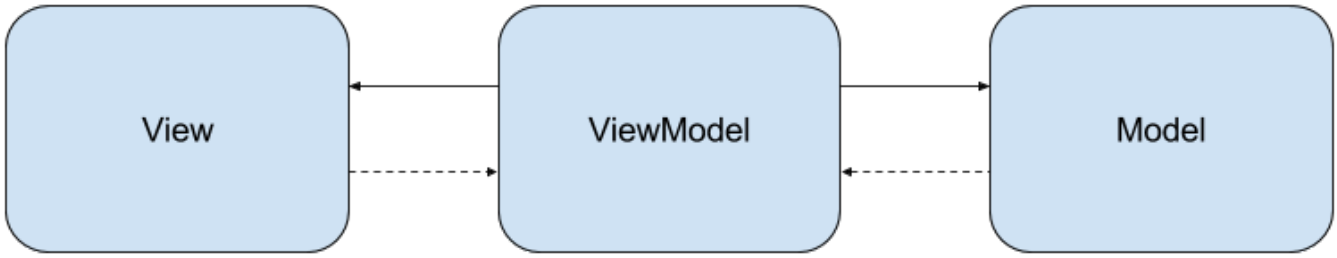
MVVM was originally developed by Microsoft in order to benefit from WPF's event driven architecture. However, on the Android platform it is possible to work with events and *observers* as well. The viewmodel is often called a *value converter* because it prepares the data from the model for the view. 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.

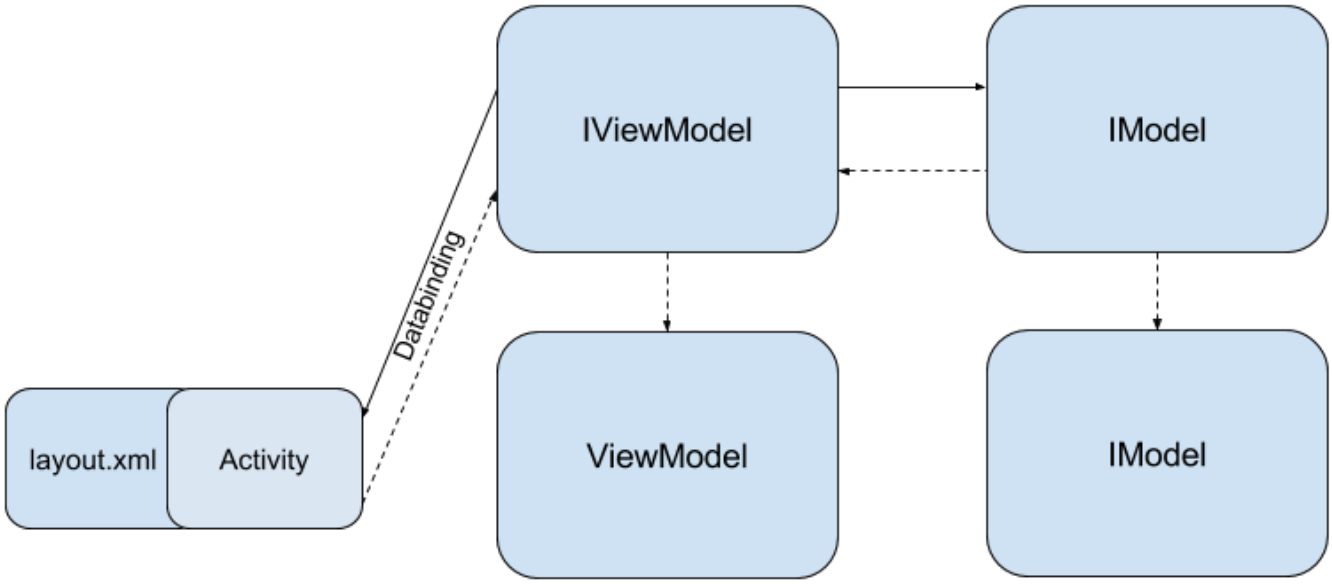
What this means is that unlike MVP, a viewmodel could potentially be shared with many views. This strict separation allows for a very decoupled architecture. Contrast this with a view-presenter relationship where the view is still responsible for manipulating information and state.(“The MVVM Pattern,”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



(18.1) 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.



(Fig, 18.2) MVVM Implementation

Flux

Action, dispatcher, store, view

- **Will be implemented**

Theory

Flux is without a doubt the newest kid on the block. It has been gaining a lot of steam as of late because it greatly simplifies how front-end code is developed in the browser. Even though Java and JavaScript can seem worlds apart, as Flux is simply a design pattern there is no valid reason not to test and validate it.

Unique to Flux, a *unidirectional* flow of data is maintained and strictly enforced not just between select components but for every component. According to the creators of Flux, this makes it far easier to reason about code structure and how data is passed around.

Flux was developed for a reason which might seem like heresy to some: any reasonably complex application built using traditional architectural patterns proved hard to maintain and extend. This problem became quite evident for Facebook when they wanted to create *instantly updated* and *highly reactive* interfaces.(“Flux Application Architecture for Building User Interfaces,” n.d.)

The pattern consist out of a few simple components:

- Dispatcher: Sends out messages when a new *action* has been received. Only one of these exists to manage a whole application

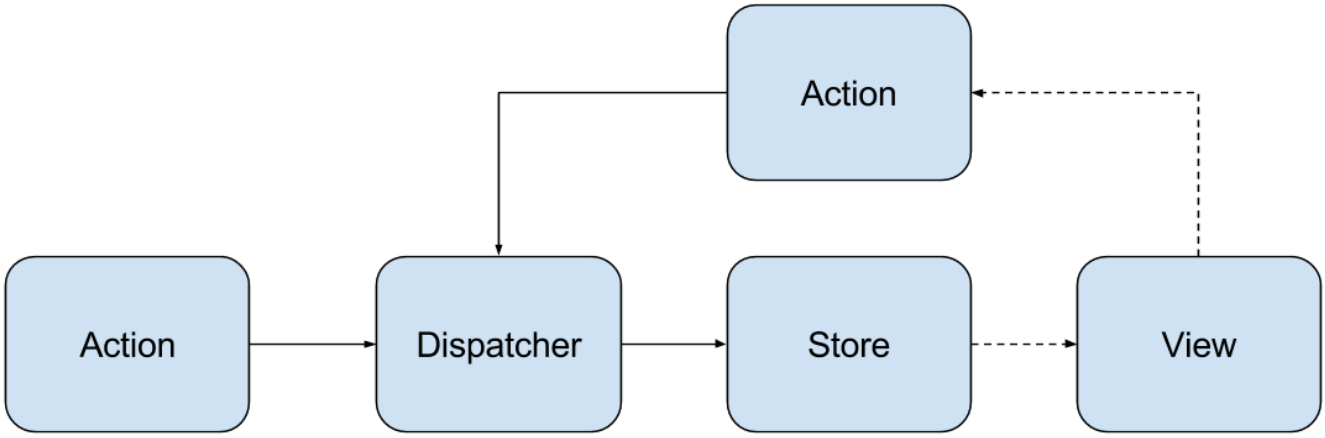
For all intents and purposes, a dispatcher is similar to a real life call dispatch: a place where customers' calls (actions) are answered, setting in motion a change that handles whatever must occur in order to handle those calls (a store).

- Stores: Manage a *domain* of state and logic

Whereas a traditional model might be responsible for a a single data object, returning it to whatever requests it, a store is the owner of a spectrum of a collection of objects.

To give an example: a *CatModel* would have a *getCat(int id)* method returning a single `cat` and a *CatStore* would have a *CatsUpdatedCallback* method which returns those cats to whichever listener is *registered* to it.(“Flux Application Architecture for Building User Interfaces,” n.d.)

- Views: Render the current application state, which is handed down from the stores
- Actions: Views send out updates and request using Actions to the centrail dispatcher



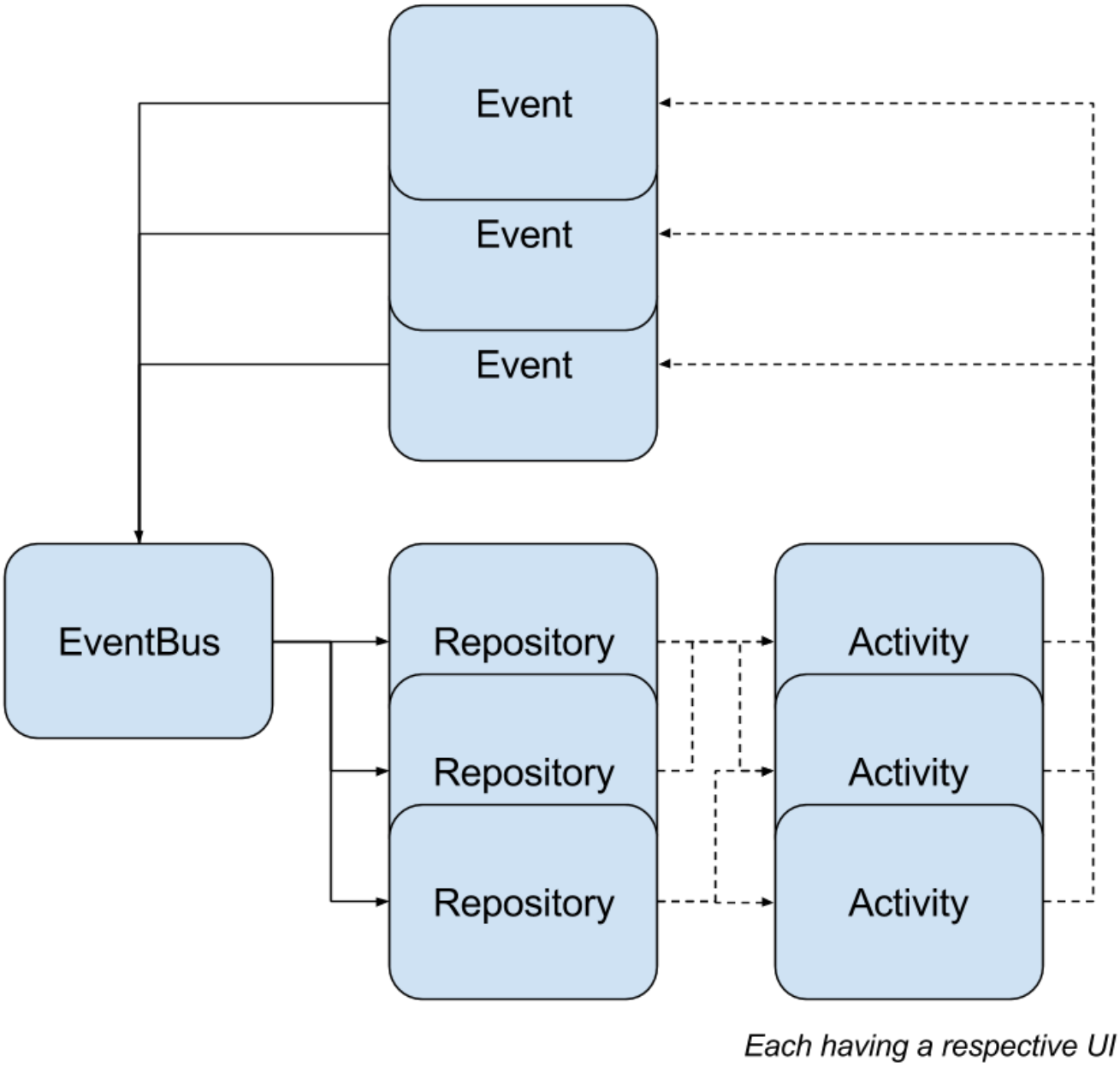
(Fig, 19.1) Flux

Implementation in Android

Despite its apparent complexity, Flux can easily be implemented in Android.

- Activities and their respective XML UI can be considered a single view component, where the Activity is set to listen to store changes and the XML is responsible for reactively responding to changes in a local *store represenation*.
- An event bus can serve as a simple dispatcher
- Stores are pure Java classes which independently decide how to parse actions and change data caches accordingly

- Actions in this interpretation are simply new events dispatched to the store



(Fig, 20.1) Flux Implementation

Note: unlike other flowcharts in this chapter, some details such as interfaces have been left out for clarity.

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

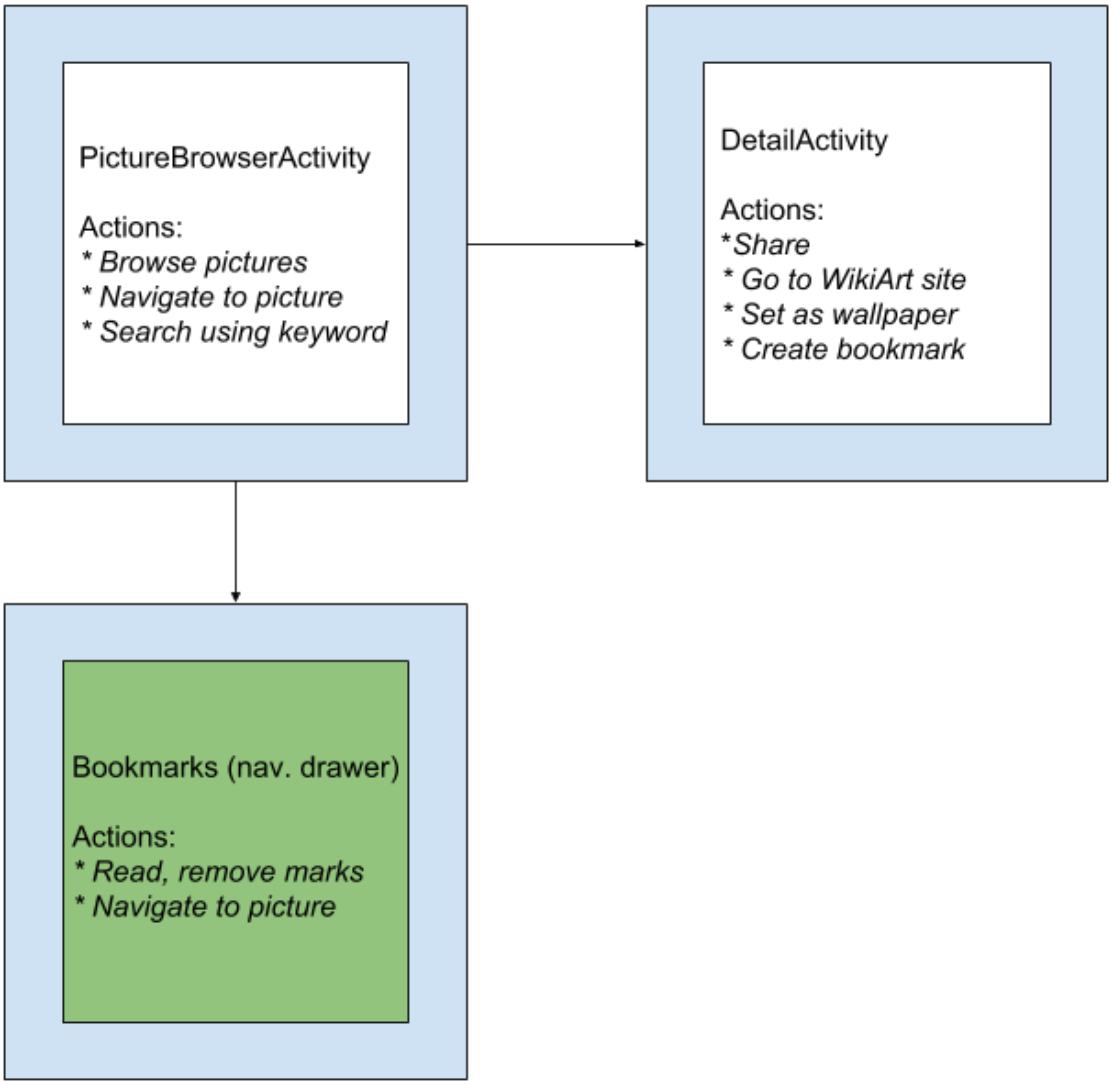
With these requirements in mind, an example app has been chosen.

The app: Tumblr Image Downloader

Tumblr is a website which on which users can host a blog. 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.



(Fig, 22.1) 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.

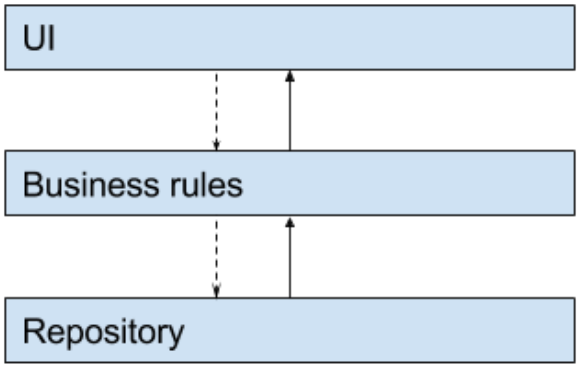
Decoupling

> Decoupling is the process of separating components so that their functionality will be more self contained. -Anders Dahnielson

This is arguably the most important metric in code architecture quality. Without a decoupled architecture, code will quickly become entangled an extremely difficult to extend. It also has a number of important consequences for how easy it is to test code and avoid duplication.

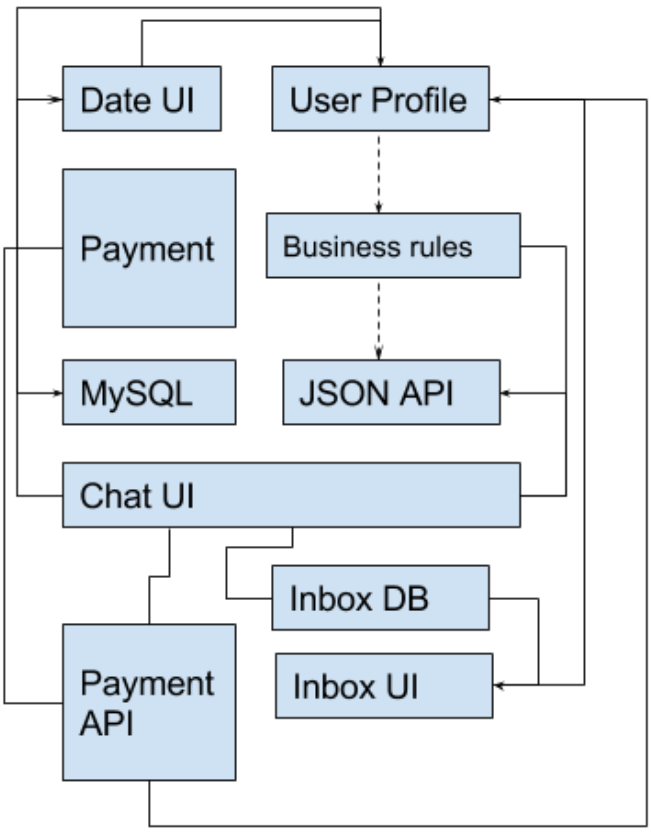
A simple test can be performed to find out how decoupled the code is: attempt to cast it into a layered diagram.

Would it be possible to describe the general architecture with just a well defined components:



(Fig, 23.1) Decoupled code

Or would it take quite some effort:



(Fig, 23.2) Entangled code

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. -Robert .v Binder

In order to make it attractive to developers to test their code, the code itself should be easy to test.

This is heavily dependent on the pureness of classes and how many side effects might occur when invoking a function. The more global state is accessed in code, the more the correct functioning of that code depends on values it can't control itself.

The relation between testability and decoupling will be examined by using Dagger in the example application.

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. -Kishor Trivedi

Fault tolerance is not just simple error handling and exception catching -it is a conscious approach to taking care of a user's data and the availability of the services an application provides.

As an example, what happens when the application loses connectivity to the internet? How much time should pass before a new attempt at connection is made? How does it affect any background processes that might need a connection?

Tolerance to unexpected circumstances is thus something that should be embedded throughout the codebase and taken into consideration from the very start of a project.

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. -MSDN, What is Extensibility?

While clearly related to the degree of decoupling, the extensibility of code is an important measurement on its own. Without a good system in place to extend existing code, it becomes increasingly difficult make additions that are clean, easy to test and not reliant on global state.

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. -Nathan Long

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.

Tools, Testing and Libraries

Before developing the application, a number of important tools and popular libraries will be reviewed. Deciding if and when a certain library should be used can have a major impact on an application. (“What is Extensibility?” n.d.) Not only does the developer have to depend on the library's maintainer to keep it up to date, without careful attention to implementation the risk of vendor lock-in becomes significant.

Android's great selection of profiling tools on the other hand, is often considered as something to use only as a last resort. Proof will be presented here that shows how invaluable they are.

Tools

The Performance Profiling Tools

Android Studio's profiling tools are spread out over several packages and inside the Developer Options menu on a phone itself.

GPU

GPU performance can be profiled in two distinct ways: on the device itself using the *GPU Overdraw* and the *GPU Rendering* tools, and using a desktop application called the *Hierarchy Viewer*.(“Why is verbosity bad for a programming language?” n.d.)

On-Device tools

The GPU overdraw tool simply displays how many times a specific part of a layout has been drawn over i.e. the amount of elements underneath a certain spot.

The level of overdraw is visualized in colors, using this scale:

(Fig, 26.1) GPU Overdraw, image property of Google

- True color: No overdraw
- Blue: Overdrawn once

- Green: Overdrawn twice
- Pink: Overdrawn three times
- Red: Overdrawn four or more times

While having no overdraw at all would be exceedingly difficult, Google generally recommends having an overdraw of at most one (blue).(“Why is verbosity bad for a programming language?” n.d.)

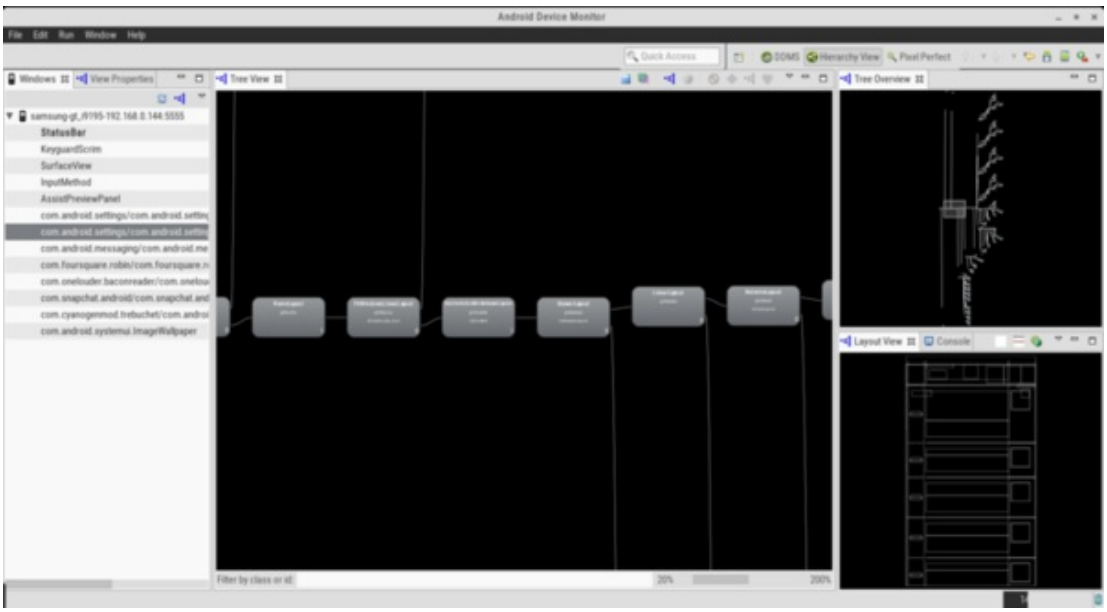
Layout Hierarchy Inspection

Every added level of GUI abstraction comes attached with added overhead. This can mean anything from choppy animations to an view taking an unacceptably long time to load.

While it might be possible to make a somewhat accurate mental visualization of a view's hierarchy, specialty tools are needed if any kind of accurate profiling is required. This is what the Hierarchy Viewer provides.(“Why is verbosity bad for a programming language?” n.d.)

The Hierarchy Viewer is divided into three main parts:

- A device list, which makes it possible to select the device that needs to be inspected.
- A console which provides device information.
- The layout and tree views, which respectively visualize the view's hierarchy and its layout in the shape of a wireframe.



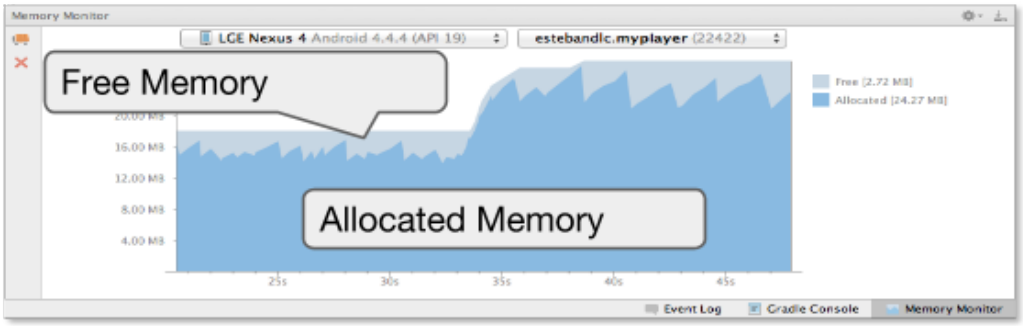
(Fig, 27.1) Easily inspecting the Settings App using Hierarchy Viewer

Memory Performance using Memory Monitor

Enabling the monitor takes a few steps:

- Make sure ADB integration is enabled by checking "Tools > Android > Enable ADB Integration"
- Open the memory tab at the bottom of the IDE window (in Android Monitor)

The monitor view should now be visible and outputting information.

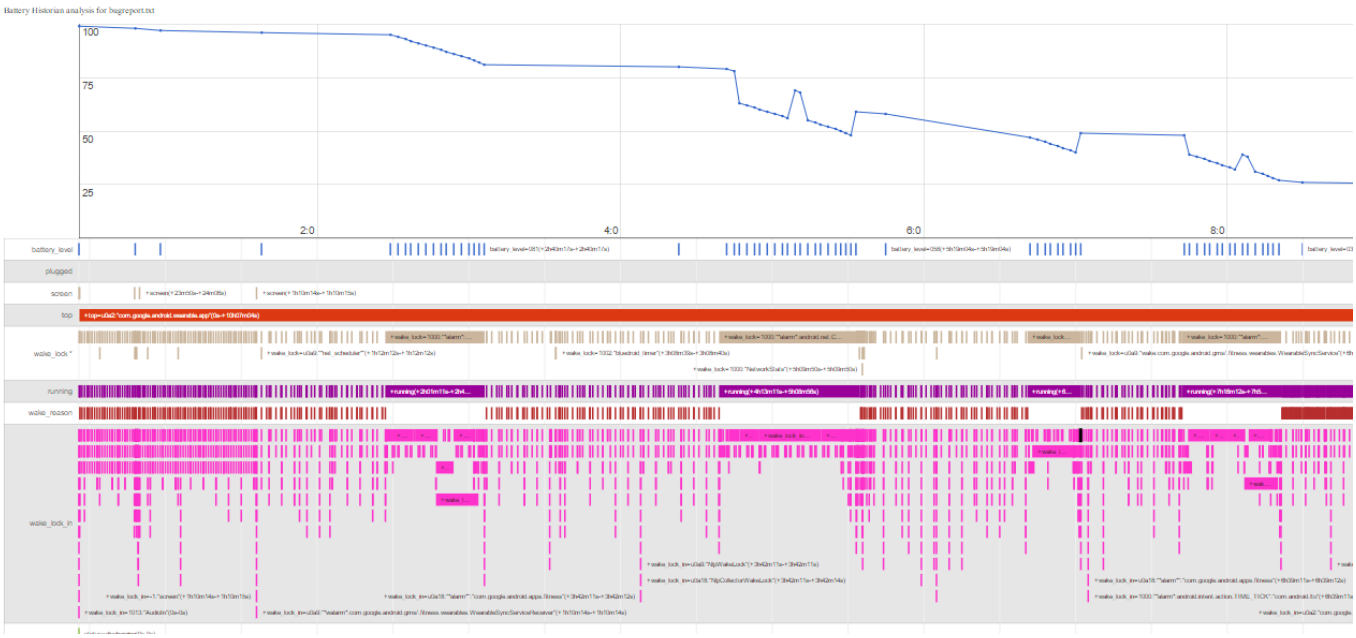


(Fig, 28.1) Memory Monitor, image copyright Google

In order to get a good overview of how much memory an application is currently consuming, the Memory Monitor tool is excellent.

In an easy to parse graph it displays the current amount of memory and the amount of free memory.(“Why is verbosity bad for a programming language?” n.d.)

Testing Battery Usage



(Fig, 28.2) A common Battery Historian view

How much battery life an application consumes should be one of the most important concerns of any mobile developer. Not only will consumers be more likely to use an application if they don't notice any considerable decrease in battery life by using it(Ferreira, Dey, & Kostakos, 2011): battery life can also give clues into how many resources every single component of an application needs.

This is tied into several other profiling tools such as the Hierarchy Viewer and the Memory Tracers, since more resources means less battery life.

Battery life is best tested using Battery Historian, an open source program published by Google. Unlike most other profiling tools, battery historian is a simple Python script and it not embedded into the Android Studio IDE.(“Why is verbosity bad for a programming language?” n.d.) This does not mean Battery Historian is difficult to work with however, as it only requires a few steps:

First, download Battery Historian from http://github.com/google/battery-historian.

Using the command line, kill the current Android Device Bridge Server:

```
> adb kill-server
```

Make sure the target device is connected:

```
> adb devices
```

Restart the battery data gathering process:

```
> adb shell dumpsys batterystats --reset
```

Next, disconnect the phone and use the app until enough data has been gathered. Disconnecting the device is crucial since otherwise it will charge its battery via USB.

Then, reconnect the device and check if it has successfully connected to ADB:

```
> adb devices
```

Dump the battery statistics to a text file:

```
> adb shell dumpsys batterystats > batterystats.txt
```

Navigate to the directory in which Battery Historian was downloaded, an issue this command:

```
> python historian.py batterystats.txt > batterystats.html
```

Network Traffic Capturing

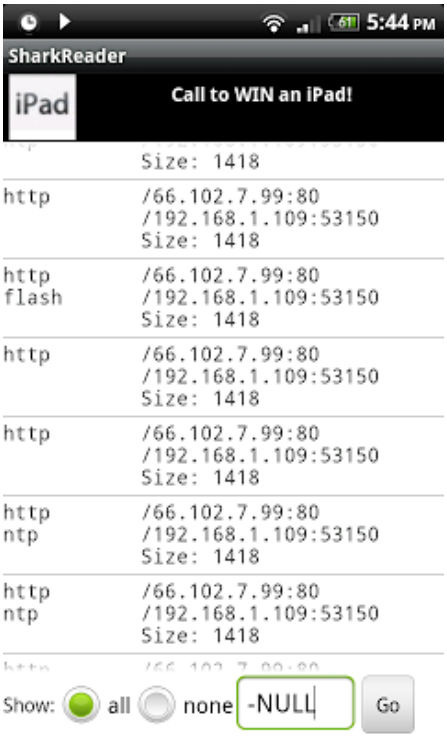
Traffic capturing is currently not very well supported by Google.

With a little hacking however, it is definitely possible.

Requirements:

- A rooted phone
- Shark for Root, an app which wraps tcpdump and can record network trafic
- Shark Reader, for viewing captures

Using this configuration it's possible to capture traffic directly on a device and view it at a later time.
There are a lot of "ifs and buts" however: most phones are not rooted and the interface is qute sluggish.



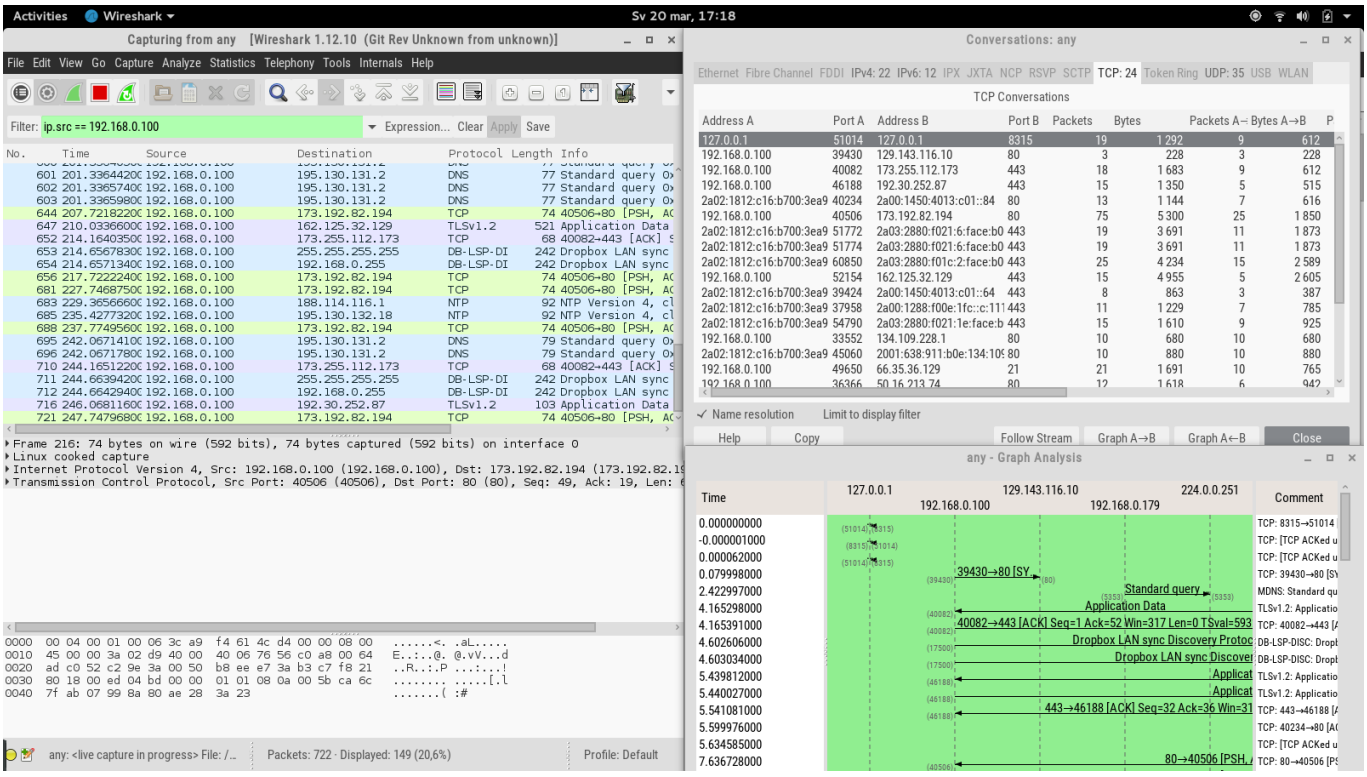
(Fig, 30.1) SharkReader: most likely not the greatest interface ever. Image copyright lanrat.com

Luckily a better alternative exists: using a computer as a wireless hotspot.

Requirements:

- A computer which can serve as a hotspot
- Wireshark/tshark

Once the device is connected to the hotspot, enter `ip.src == *Device IP*` in Wireshark's filter bar. This will show every package orginating from the phone in question.



(Fig, 30.2) Viewing TCP conversations and a TCP flow graph of a connected device

By using the `tshark` command in the console, it's also quite easy to directly view GET requests.

```
# anthony at madhvani in ~/Dropbox/Bachelorproef on git:master x [17:22:16]
$ sudo tshark -i any \
            -Y 'http.request.method == "GET"' \
            -T fields \
            -e http.request.method -e http.request.uri -e ip.dst
Running as user "root" and group "root". This could be dangerous.
Capturing on 'any'
GET     /blog/2016/03/16/tcpdump-is-amazing/
GET     /stylesheets/screen.css
GET     /css?family=PT+Serif:regular,italic,bold,bolditalic
GET     /cdn-cgi/nexp/dok3v=e982913d31/cloudflare.min.js
GET     /css?family=PT+Sans:regular,italic,bold,bolditalic
GET     /css?family=Alegreya:400,900,700
```

(Fig, 31.1) Viewing GET requests using tshark

It should also be noted that tshark has deep knowledge of a very large amount of TCP protocols and not just HTTP, like for example database-specific MongoDB TCP calls.(Evans, n.d.)

This technology stack should provide sufficient tooling for almost all situations.

Testing

This chapter will briefly detail the testing frameworks that will be utilized in testing the functionality of the example application. Proficiency in testing is already assumed on the reader's part as an introduction to testing itself it out of the scope of this thesis.

Several great resources are available that serve as an excellent introduction to testing and test driven development, such as *Test Driven Development: By Example* and *Pragmatic Unit Testing in Java 8 with JUnit*.

Robolectric

> Robolectric is a unit test framework that de-fangs the Android SDK jar so you can test-drive the development of your Android app.

Robolectric forms the glue between unit testing and the Android SDK. This library makes it possible to efficiently test an application's UI code in isolation from its business logic. Separating UI and business logic is achieved primarily by way of *mocking* out (also known as *faking*) most other code.(“Robolectric,” n.d.)

This thesis's source code, for example, was tested using Robolectric because it has several major benefits over Espresso:

- Nearly the whole Android SDK can be faked, while providing features those libraries normally would not have such as internal inspection and reflection.
- Excellent support for multi-threaded classes such as AsyncTasks and Handlers. This works wonderfully for testing the return value of functions off the main thread and inspecting them.
- Non-UI code does not need to be tested on an emulator, which can tremendously speed up the test-develop-test cycle.
- Add-ons are available which provide testing support for Google Play Services (among others).
- Activity creation can be mocked out and controlled via an `ActivityController`, with `Fragment` and `View` mocking functioning in a similar manner on an `ActivityController` as well.

("Robolectric vs Android Test Framework - Stack Overflow," n.d.)

The basic workings of Robolectric are explained no better than by the authors themselves:

```
@RunWith(RobolectricTestRunner.class)
public class MyActivityTest {

@Test
public void clickingButton_shouldChangeResultsViewText() throws Exception
{
        MyActivity activity = Robolectric.setupActivity(MyActivity.class);

        Button button = (Button) activity.findViewById(R.id.button);
        TextView results = (TextView) activity.findViewById(R.id.results);

        button.performClick();
        assertThat(results.getText().toString()).isEqualTo("Robolectric Ro
cks!");
    }
}
```

First, through a Testrunner annotation, it is declared that this test should be performed using Robolectric. Using this identifier tells the compiler that this is an actual Robolectric test and so it is required.

Next, a new activity is instantiated (and its XML inflated) using *setupActivity* by providing that method with a class deriving from *Activity*.

Following that a `Button` and a `TextView` are instantiated from the Activity's associated XML.

Finally, a button click is performed which the test expects to make `results` text say "Robolectric Rocks!"

Besides a single Robolectric-specific Activity instantiation and an assertion at the end, this unit test is exactly the same as any regular Android code. The unity between test code and actual code makes writing Robolectric tests fairly straightforward (which is exactly what the authors intended).

Libraries

Dagger2

Dagger2 is considered the standard dependency injection (inversion of control) library for Android. Dagger was originally developed by Square and subsequently forked by Google.

Dagger was created to get all the benefits of dependency injection without the boilerplate. ("Dagger ‡ A

fast dependency injector for Android and Java.” n.d.)

Reasons for using a dependency injection framework on Android:

- Easily swap out dependencies with fakes
- Make different configurations by simply changing how a dependency is resolved
- It's declaratice: there is no logic involved in setting up the dependencies
- Error reports and graph composition at compile time
- No reflection overhead
- Sane debugging with few stack frames and human-friendly generated class names

Dagger is essentially made up out of several annotations which tell it how dependencies should be resolved:

- `@Inject`
- `@Module`
- `@Provides`

@Inject is used to annotate classes which Dagger should be allowed to instantiate. It may also be used on fields in order to tell Dagger how if those should be resolved as well.

```
//Code taken verbatim from http://google.github.io/dagger/users-guide
class Thermosiphon implements Pump {
  private final Heater heater;
  @Inject Pump pump;

  @Inject
  Thermosiphon(Heater heater) {
    this.heater = heater;
  }
}
```

In this class, `heater` and `pump` never need to be instantiated as it will be resolved by Dagger.

Sometimes a simple inject does not suffice, for example when using a third-party library or when code needs to be configured. That's where @Provides comes in.

```
//Code taken verbatim from http://google.github.io/dagger/users-guide
@Module
class DripCoffeeModule {
  @Provides static Heater provideHeater() {
    return new ElectricHeater();
  }

  @Provides static Pump providePump(Thermosiphon pump) {
    return pump;
  }
}
```

Here, it is declared that a `Heater` should resolve to `ElectricHeater` and a `Pump` to a `Thermosiphon`. Note that providePump itself also has a dependency.

```
//Code taken verbatim from http://google.github.io/dagger/users-guide
@Component(modules = DripCoffeeModule.class)
interface CoffeeShop {
  CoffeeMaker maker();
}

CoffeeShop coffeeShop = DaggerCoffeeShop.builder()
    .dripCoffeeModule(new DripCoffeeModule())
    .build()
```

Having built up the dependency graph, the dependency injection can be instantiated by using Dagger's generated code (which it made by examining the `CoffeeShop` interface).(Google Developers, 2014)

RxJava

RxJava and RxAndroid are the standard libraries for making *Reactive Extensions* available to Android. What it essentially does is make it far easier to write asynchronous code that using event-based observables.

Reactive Extensions exist out of two basic building blocks: *Observers* and *Observables*. Observables emit a value which an Observer can receive by subscribing to an Observer.(Dupree, n.d.)

Here is what typical RxJava code looks like, in order to display weather events from a fictitious forecasting service:

```
Observable<List<Forecast>> weatherObservable = Observable.fromCallable(new
Callable<List<Forecast>>() {
//The network call needed to receive a weather forecast is placed in a Cal
lable, which makes it non-blocking

    @Override
    public List<Forecast> call() {
    //call() will be called when an Observer subscribes
        return httpClient.getForecasts("today");
    }
});

weatherSubscriber = weatherObservable
    .subscribeOn(Schedulers.io()) //The Observable is set to run on a sepa
rate thread, so the main UI thread is never blocked
    .observeOn(AndroidSchedulers.mainThread()) //The Observer is set to re
ceive events on the main UI thread
    .subscribe(new Observer<List<Forecast>>() {

        @Override
        public void onCompleted() { } //This event is fired when the Obser
vable is done emitting events

        @Override
        public void onError(Throwable e) { } //If an error occurs, it will
be thrown here

        @Override
        public void onNext(List<Forecast> weather){ //When a new value is
received from the Observable's stream, it will be sent here
            displayWeather(weather);
        }
    });
```

Note that a different type of Observable called a Single also exists, which only emits two events: onCompleted and onNext. In this use case it would be preferable since only one event is ever emitted.

The RxJava standard library is positively huge and almost every possible use case has been accounted for.(“ReactiveX - Operators,” n.d.)

Several other packages also exist which extend RxAndroid such as*RxLifecycles*, which helps with unsubscribing from observables to kill of any possible remaining threads.

Mortar

> A simple library that makes it easy to pair thin views with dedicated controllers, isolated from most of the vagaries of the Activity life cycle.

- **Mortar will be used in the MVP version of the example app, so only a cursory view explaining the reasoning behind Mortar is provided here.**

Mortar was developed to make it easier to use a `View` as the basic unit of an Android application, as opposed to fragments. The reasoning behind this is that Fragments introduce a huge amount of complexity to the flow of an application without adding many benefits.

The functionality Mortar provides essentially simplifies access to an Activity's lifecycle events using its `BundleService`. Besides this, it also contains a `Presenter` class which builds upon this service, meant as an aid in developing applications in an MVP pattern.(“Square/mortar,” n.d.)

Why forego Fragments?

In a well-known post on Square's Engineering Blog, a part-rant part-solution was provided for managing an app in a well-structured way.

In this blog, the combination of Fragments' and Activities' lifecycle was dubbed the "lolcycle". Besides being complex in usage, it also makes Android extremely hard to debug.

So an alternative was provided to the lolcycle: using custom views. Sharing the lifecycle of Activities with those views made it far easier to develop but an other problem became obvious: there was a need for decoupling UI from logic.

This eventually resulted in a combination of custom views needing access to lifecycle events and presenters controlling those customs views.

As a solution to this Mortar was developed(“Advocating Against Android Fragments,” n.d.)

Building the Example App

This chapter documents how the example application was developed using various architectures. Besides a working implementation, experiences and conclusions are documented so as to provide guidance to developers in choosing a general structure for their application.

Readers are encouraged to follow along and develop a simple application as well.

To reiterate, the application which will be built is a simple API client which:

- Must make a number of network calls (the model)
- Must convert that data into a format suitable for end-users (the controller/presenter/viewmodel/store)
- Must present that data to the user (the view)

On each implementation, a specific library will also be showcased which is both widely-used and vastly simplifies development. This to combat the common way of presenting a design pattern: without using any "helper" libraries, resulting in code that's quite contrived and hard to understand.

Unlike most projects classes were packaged in order to decrease inter-package coupling, an important consideration on its own.(Sandin, 2016) Contrast to packaging classes by *Kind*, e.g. placing all Activities in one package.

MVP using Mortar

```
├── Repositories
│   └── Blogs.java
├── PictureBrowser
│   ├── PictureBrowserAdapter.java
│   ├── PictureBrowserPresenter.java
│   ├── PictureBrowserView.java
│   └── PictureBrowserActivity.java
├── PictureDetail
│   ├── PictureDetailActivity.java
│   ├── WallpaperImageView.java
│   └── PictureDetailPresenter.java
```

(“MVP for Android,” 2014–2014-04-15T15:19:55+00:00)

Project setup

First of all, add the Mortar dependency to the application's Gradle file:

```
compile 'com.squareup.mortar:mortar:(latest version)'
```

Development

Tip: Read the Docs. Mortar is a fully fledged library so quite some knowledge about its workings is required in order to use it efficiently.

Creating the MortarScope

The MortarScope is a singleton instance in a subclassed application:

```
package thesis.madhvani.tk.artbrowser_mvvm

import android.app.Application;
import mortar.MortarScope;

public class ArtApplication extends Application {
  private MortarScope rootScope;

  @Override public Object getSystemService(String name) {
    if (rootScope == null) rootScope = MortarScope.buildRootScope().build(
"Root");

    return rootScope.hasService(name) ? rootScope.getService(name) : super
.getSystemService(name);
  }
}
```

This exposes the *Scope service*: an object which can build child scopes. Among other things these objects make it possible to reference Presenters.

View and Activity

As previously mentioned, Mortar is based on extending views instead of using fragments as a method of organizing code.

It's important to remember that Mortar is not an absolute requirement for this, it just makes the process far easier.

In order to implement a Mortar compatible View class, some steps must be undertaken:

- Get a reference to the relevant presenter using the MortarScope
- Override some base lifecycle methods Views have, and detach/attach the presenter from it
- Create references to each field in the UI

  public class WallpaperImageView extends RelativeLayout { public PictureDetailView(Context context, AttributeSet attrs) { super(context, attrs); ObjectGraphService.inject(context, this); } }

Displayed here is the minimum amount of code required to have a working custom view. Unlike most of the Mortar framework, this type of subclassing does not require any external libraries. Even without using any specialized packages custom views are still an important method of encapsulating behavior. Mortar is worthy of using because it enables those custom views to integrate very well in the larger contect of an application and not the other way around.

Layout files

Since Mortar uses extends views, the layout becomes very declarative:

```
<thesis.madhvani.tk.artbrowser_mvvm.WallpaperImageView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center"
    >

  <ImageView
      android:id="@+id/image"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:textSize="25sp"
      android:gravity="center"
      android:text="WTF!?"
      />
  <Button
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="Rotate me and watch the update count increase."
      android:textSize="18sp"
      android:gravity="center"
      tools:ignore="HardcodedText"
      />
</thesis.madhvani.tk.artbrowser_mvvm.WallpaperImageView>
```

The only noticeable difference is the usage of custom views, which will be explained shortly.

Presenter

```
class PictureDetailPresenter extends ViewPresenter<WallpaperImageView> {
  private final DateFormat format = new SimpleDateFormat();
  private string url;

  public PictureDetailPresenter(String url){
    this.url = url;
  }

  @Override protected void onLoad(Bundle savedInstanceState) {
    getView().setImage(getImage(url));
  }

}
```

This is a basic form of a Mortar "ViewPresenter", which for all intents and purposes is just a regular Presenter with a few bells and whistles. What is very pleasing about using Mortar is that Presenters can be used very cleanly, i.e. without much programmer interaction needed in order to fetch and save state. But although the library's features allow for very smooth development, some overhead is still associated with using it. In some cases this can be safely ignored and in other, bigger applications, a custom MVP implementation might be needed. **Always consider the specific needs of every individual project.**

Saving state in the presenter

Without Mortar's wrappings, the question will come on how to save a presenter into an activity. A simple solution is presented here which can easily be modified.

Simply saving a presenter's state in a `Map<id, singleton>` will suffice in most cases. This makes it possible to fetch any existing presenter during the `onCreate` call. This is essentially also what Nucleus does, another helper library for using MVP on Android.

Conclusion

MVVM using the Data Binding library

```
├── Repositories
│   └── Blogs.java
├── PictureBrowser
│   ├── PictureBrowserAdapter.java
│   ├── PictureBrowserViewModel.java
│   └── PictureBrowserActivity.java
├── PictureDetail
│   ├── PictureDetailActivity.java
│   └── PictureDetailModel.java
```

Project setup

As the Data Binding Library recently had its first stable version released, it can safely be used without having to worry about too many unexpected glitches. Another thing to point out is that it's a support library, meaning all versions of Android after API level 7+ support it.

The only requirement on the developer's side is that a relatively recent version of Android Studio is needed (1.3+) in order to have syntax highlighting and an error catching.

First, enable the library by pasting the following code into an application's Gradle file:

```
android {
    dataBinding {
        enabled = true
    }
}
```

Development: Understanding Data Binding

View

First, the view will be examined. The Data Binding Library is best understood by looking at a real world example.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
      <variable name="image" type="thesis.madhvani.tk.artbrowser_mvvm.pict
ure.PictureViewModel"/>
      <import type="android.view.View"/>
    </data>
    <RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context="thesis.madhvani.tk.artbrowser_mvvm.PictureDetailAct
ivity">

        <ProgressBar
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          android:visibility="@{image.isLoaded ? View.GONE : View.VISIBLE}
"/>

        <thesis.madhvani.tk.artbrowser_mvvm.TargetedImageView
            app:imageSource="@{image.url}"
            android:visibility="@{image.isLoaded ? View.VISIBLE : View.GON
E}
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_alignParentTop="true"
            android:layout_alignParentStart="true"
            android:layout_alignParentBottom="true"
            android:layout_alignParentEnd="true"/>

        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{image.caption}"/>

    </RelativeLayout>
</layout>
```

Without attention this XML file could be mistaken for any regular RelativeLayout. There are however some crucial differences:

First of all, the XMl is wrapped inside a `<layout>` tag and some metadata is present in the file:

- `Data`: this is a special XML tag in which models and imported classes need to be declared. In the example application, a `PictureViewModel` has been set as the main view model (more on the view model later). An import, namely `<import type="android.view.View"/>` is also declared. This makes it possible to easily reference classes inside the view.

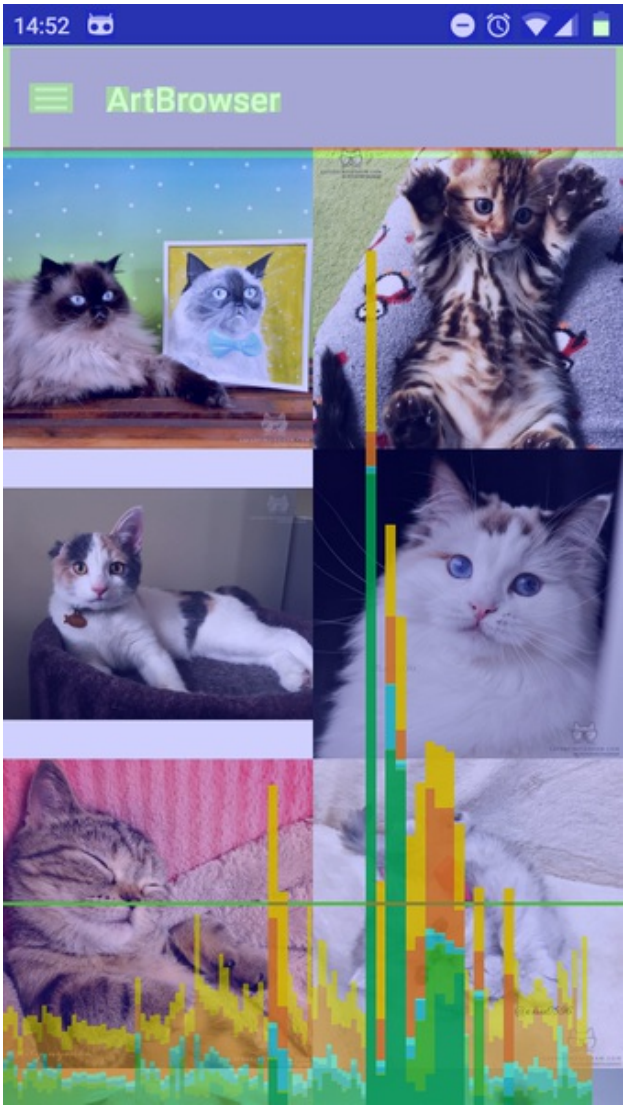
The actual UI markup also has some special characteristics:

- `android:text="@{image.caption}"`: developers who have previous experience with templating languages such as Razor or Mustache will be instantly understand what this statement means. It simply sets the text value to a String fetched from the model.

- `app:imageSource="@{image.url}"`: this statement makes a references to a *BindingAdapter*, a custom method inside `PictureViewModel` code that essentially makes it possible to get a reference to the `View` widget that calls it. In this case, the "callback" is used to set a `BitmapImage` loaded from an API. Note that the custom `View` object `thesis.madhvani.tk.artbrowser_mvvm.TargetedImageView` is not necessary in order to use a `BindingAdapter`, this is related to image caching and can be ignored.

- `android:visibility="@{image.isLoaded ? View.VISIBLE : View.GONE}`: the library also has support for arbitrary expressions as shown here. Without any code-behind, a progress bar can be shown as the image is loading and removed as soon as it the image is available.

The best part about understanding Android's Data Binding Library and how it relates to views is that as a consequence the essence of a view in MVVM on Android also becomes clear: simple, easy to understand XML files which will eagerly accept and display any data handed to them.(“Data Binding Guide Android Developers,” n.d.)



(Fig, 44.1) Despite little overdraw, a performance hit was noticeable

ViewModel and Activity Initialization

On Android, delegating all responsibility to the ViewModel would be very impractical since an Activity object is responsible for holding on to state, the initial inflation of the interface and a number of other administrative tasks such as handling rotations.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   PictureDetailActivityBinding binding = DataBindingUtil.setContentView(t
his, R.layout.picture_detail_view);
   PictureViewModel picture = new PictureViewModel("http://i.imgur.com/wUu
wrjy.jpg", "The cat of our vet...", "Saturnix, Reddit");
   binding.setImage(picture);
}
```

Note that some code has been left out of this example (such as fetching a Picture object from an `Intent`) for clarity.

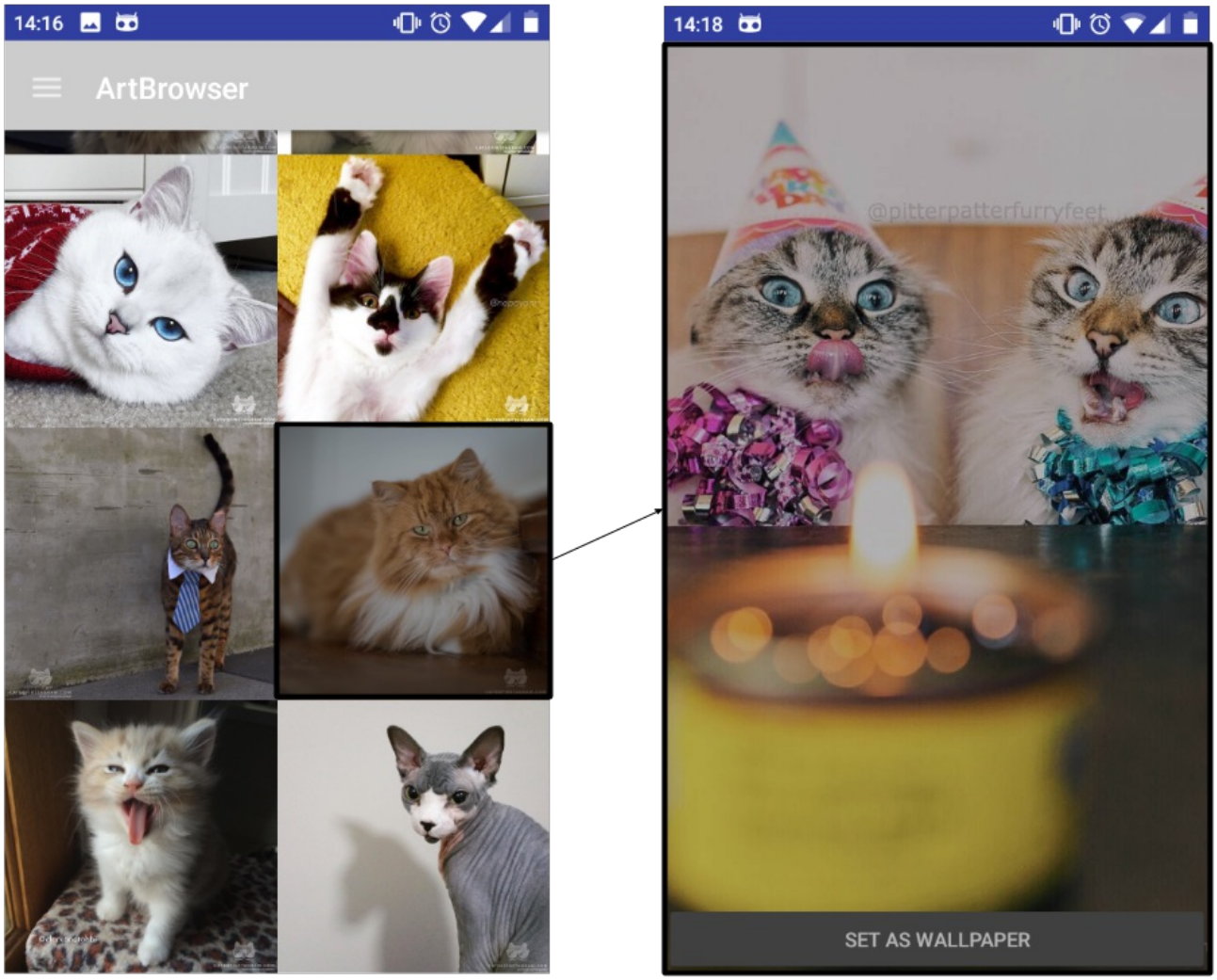
Possibly the most perplexing statement in this onCreate method is the one that defines the binding. `PictureDetailActivityBinding` is actually just the name of the Activity set in PascalCase and appended with "Binding". This class is automatically generated when an XML file that uses databinding is built.(“Data Binding Guide Android Developers,” n.d.)

Afterwards the `Image` variable declared in the View is simply instantiated.

```
public final class PictureViewModel extends BaseObservable {

    private Image image;
     public ObservableBoolean isLoaded = new ObservableBoolean(false);

    public PictureViewModel(String url, String caption, String author) {
        this.image = new Image(url, caption, author)
    }

    public PictureViewModel(Image image) {
        this.image = image;
    }

    public String getUrl() {
      return image.getUrl();
    }

    public String getCaption() {
      return image.getAuthor() + ": " + image.getCaption();
    }

    @BindingAdapter({"bind:imageSource"})
    public static void loadImage(ImageView view, String url) {
        image.loadBitmapImage(url)
        //setImageBitmap would normally happen in a callback once loadBitm
apImage is done
        view.setImageBitmap(image.getBitmapImage());
        isLoaded.set(true);
    }

}
```

- `BaseObservable` is a *convenience* class that implements an Obverser pattern, allowing the viewmodel to notify the view of any changes.

- An Obversable boolean `isLoaded` is set to `true` when the bitmap is loaded. Upon the value changing, the view will be notified and turn off the Progress View.
- The `@BindingAdapter` annotation on `loadImage` tells Android that this method is responsible for answering any calls on a custom attribute (in this case, imageSource). The View is passed in and the image is set accordingly. Besides being an easy way to set values for which there is no XML equivalent, it also encapsulates view/viewmodel communication in a well defined space.
- Special attention should be paid to `getCaption()`: a field call from the model not available from the view is called (`getAuthor()`), decreasing the knowledge the view has which keeps it "dumb" as intended.



(Fig, 46.1) Image caching logic is shared by using the same viewmodel, only the view is different here

Conclusion

Because of the Data Binding Library, implementing an MVVM system was quite easy. Its event driven nature fits in perfectly with the pattern.

Another important finding was that, even though the library is fairly new, in terms of stability there were no major issues. It can therefore be safely recommended.

One major advice is that performance sensitive UI should be inspected to make sure the performance decrease is still acceptable. Frame skipping was especially noticeable when loading a big collection-

based view such as a `RecyclerView`.

This is despite Google's best efforst at ensuring smooth performance. As it is, the library uses no runtime reflection at all: every view reference is predetermined at compile time. Also, since there is certainty about which variables a view wants access to at compile time optimizations can be made that make data access less expensive.

Just as the pattern promises, the viewmodels proved to be quite easy to reuse in different environments. An excellent example of this is the PictureViewModel being used on its own in the `PictureDetailActivity` and as a single item of a larger collection in the `PictureBrowserActivity`.

Besides performance, MVVM turned out to have a high degree of*developer happiness*. The least amount of code possible is spent on tasks such as referencing views or patching viewmodels to make them compatible with multiple activities.(“What is Extensibility?” n.d.)

In conclusion, the MVVM pattern is easy to implement and fits the Android model of code very well. Its encapsulation of data in viewmodels and the high degree of decoupling between all components made rapidly creating an application a pleasant experience.

Flux using EventBus

```
├── Actions
│   ├── CurrentBlogChanged.java
│   └── PictureStoreChangedEvent.java
├── PictureBrowser
│   ├── PictureBrowserAdapter.java
│   └── PictureBrowserView.java
├── PictureDetail
│   └── PictureDetailView.java
└── Stores
    └── PictureStore.java
```

Project setup

Add EventBus and RxJava to the project's Gradle file:

```
compile 'org.greenrobot:eventbus:3.0.0'

compile 'io.reactivex:rxjava:recent-version'
```

Development

Setting up stores

Stores are responsible for registering themselves with the dispatcher. Using events, an observer pattern will be used that automatically reacts to dispatched actions.

```
public class PictureStoreChangedEvent { }
```

Stores must also be initialized of course. A choice was made to simply create a collection of singletons and initialize them in a subclassed Application. This is maintainable up to a point but as the amount of stores grows they should be created using a dependency injection library such as Dagger2.

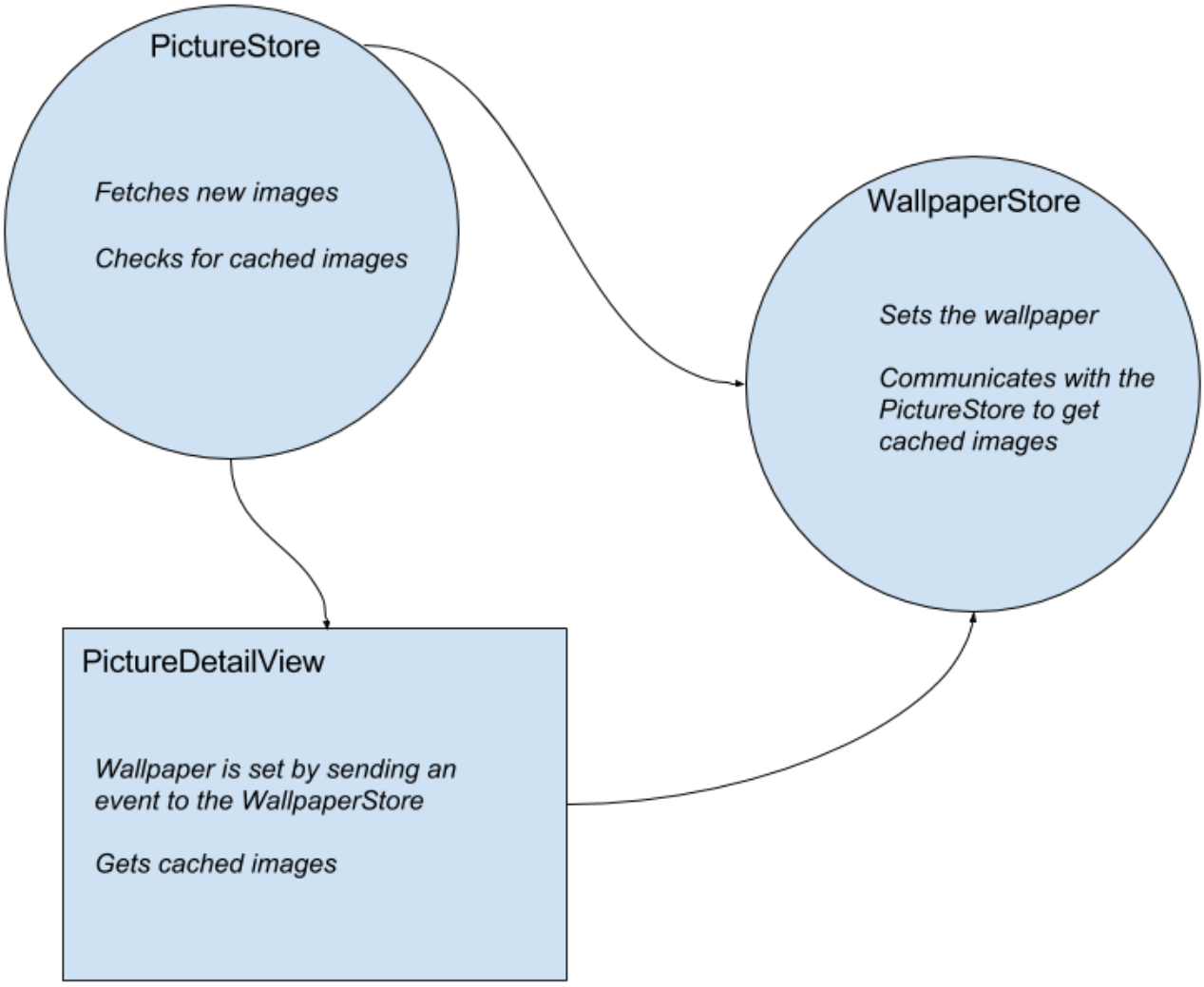
How stores actually maintain and update information is highly dependent on application specifics. It would be entirely acceptable to update something in a file, fetch data from the internet and write to an SQL database in the exact same store. This is what Flux's creators mean by maintaining a *domain of information*. What the concrete data looks like doesn't matter.

Activities and Fragments then listen to stores changes by registering to a store event.

```
@Subscribe
public void onPictureStoreChanged(PictureStoreChangedEvent event){
    catPictures = pictureStore.getCatPictures();
    //The "view" decides on its own how it responds to changes
}
```

Subscribers must also register themselves to the `EventBus`. As per the developers of EventBus, this should happen in the `onStart` and `onStop` events of Android lifecycle classes.

```
@Override
public void onStart() {
    super.onStart();
    EventBus.getDefault().register(this);
}

@Override
public void onStop() {
   EventBus.getDefault().unregister(this);
    super.onStop();
}
```



(Fig, 49.1) Thinking in React: high level Store/View communication flow (dispatcher not pictures)

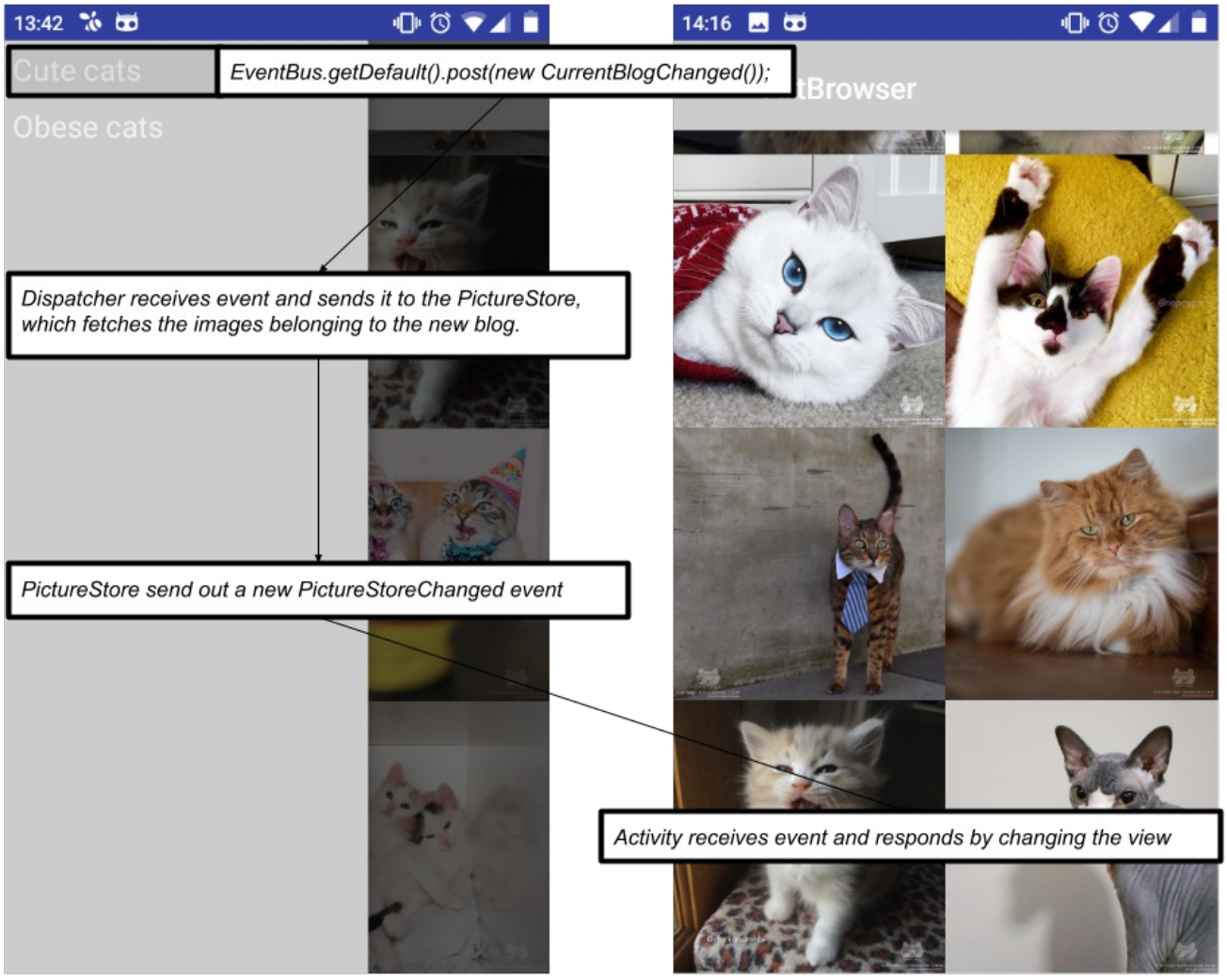
Setting up actions

Actions are a collection of events that send new state and data to the central dispatcher.

```
public class CurrentBlogChanged {

  public final String message;

  public CurrentBlogChanged(String message) {
      this.message = message;
  }
}
```

This event is then dispatched to any store that is interested in this type.

```
@Subscribe
public void handleBlogChange(CurrentBlogChanged event){
    getPictures(event.message);
    // ... And it sends out a PictureStoreChangedEvent
    // when it has fetched to notify any listeners
    EventBus.getDefault().post(new PictureStoreChangedEvent());
}
```



(Fig, 50.2) Flux's flow in practice

Deciding which type of view should be used

Flux expects view to be highly *reactive*. As a matter of fact, Flux was made with the JavaScript React.js framework in mind. How React.js functions will not be discussed here but suffice to say it should be thought of as databinding on steroids.(“Facebook,” n.d.)

A solid recommendation would therefore be to implement the Data Binding Library here as well, with `Observable<T>` fields used for maintaining state.

Conclusion

Implementing Flux was relatively painless. Most friction came from the fact that it's a completely foreign and unexplored environment for the framework, so pre-existing information was close to zero.

Another point is that Flux did not really shine that much considering the application flow was quite simple and easy to handle. Whereas *one* store was used and only a few actions, Flux is intended to be used with *many* stores, actions and views. However, what little state there was to maintain, Flux took care of without any issues.

The reactive nature of Flux and RxJava's flow control could potentially form a great combination. This is left as an exercise for the reader.

Consider an activity which contains many custom View elements, each having their own Flux View: each component would be responsible for its own state and communicating any requested changes. Thinking in Flux could then provide huge benefits as no coupling would exist between those solitary views. A possible follow-up project could be creating a more complex application to test Flux to its true limits.

Medium to large applications, which have lots of user interaction and state that needs to instantly update, could definitely benefit from using this pattern.

Concluding Analysis of 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

MVP performance was expectedly great, it has a lot of developer support from the community so most edge cases have a drop-in solution ready. As a result of this community backing, MVP would be a great choice when performance is an important issue.

MVVM

MVVM performance tends to vary wildly between how it's applied. RecyclerViews took a small hit but other, more static views actually faired better.

Flux

Flux's resource usage is mostly dependent on its use of an event bus and how that event bus interops with Android. In limited testing, performance was acceptable. The question remains wether this would still be the case in larger codebases.

Android lifecycle management

MVP and MVVM

Both patterns use a structure that require lifecycle classes to dispatch their events to an intermediary class, be it a viewmodel or presenter. This mismatch between the Android framework and the architectural pattern can be solved by way of some indirection (retrieving the object during `onCreate` by storing it in a `map` beforehand or using Mortar's BindingScope for example). So while the lifecycle can be managed, it requires extra work.

Flux

Using the Flux architecture, Activities and Fragments still have full authority over how the lifecycle is manipulated. This one to one mapping with the standard way of development means the lifecycle can be managed very well in Flux.

Testability and Debugging

When using a well-defined architecture

Every architecture mentioned so far had testing in mind when it was developed. This means that UI, business logic and models can be tested independently from each other which results in pure tests that do not need to consider other components.

A possible edge case does exist for Flux: its high degree of decoupling might make it difficult to trace program execution for debugging.(C. Martin, n.d.)

When not

By not using a structure, a developer creates an environment in which testing will become messy fast.

This is better known as the "spaghetti code" antipattern(“Design Patterns and Refactoring,” n.d.)

Take for example a Fragment that also includes a method for making API calls: it would be near impossible to test the network call in isolation since it will be coupled to the Fragment's context and lifecycle.

Because of this it is strongly discouraged to develop applications in an ad-hoc way or with strong coupling between separate functionality.

Decoupling

MVP

While MVP is a step up from MVC, view-presenter coupling is still quite severe since there should nominally be a one-on-one mapping between view and presenter. This is unfortunate in an environment such as Android where a view and its functionality might have to be reused many times over in slightly different contexts.(Richards, n.d.)

MVVM

MVVM is quite a lot more decoupled from the View than its MVP counterpart. This is a result from the philosophy behind MVVM that says views should simply accept and display information. Because of this, MVVM feels like a more natural fit for Android than MVP. Although this might sound controversial considering the amount of traction behind MVP, MVVM deserves at least an equal amount of attention.

Flux

Flux is entirely decoupled because all inter-component communication happens through an event-based system. A component does not have any knowledge about other components at all. The only coupling present is between a Store and a View: the view must know how to retrieve data from the store. By using a well thought-out interface this would be solved as well, however.

Verbosity

As was expected, all patterns came attached with extraneous boilerplate code. This is mostly because of Android's old Java version and its shortcomings as a language. Luckily improvement is on the way: at the most recent Google Developer Conference an announcement was made that many Java 8 features would be ported to Android's Jack compiler, including but not limited to: lambdas, streams and functional interfaces.(“Marshmallow Brings Data Bindings to Android, with Yigit Boyar and George Mount,” n.d.)

Overview

Architecture	Performance	Lifecycle management	Testability	Decoupling	Verbosity
MVP Pattern	Great	Acceptable	Acceptable	Quite poor	Quite poor
MVVM Pattern	Acceptable	Acceptable	Great	Great	Quite poor
Flux Pattern	Great	Great	Acceptable	Great	Quite poor

(Table,54.1)

Conclusion

Choosing the correct architecture can be a daunting task. It's obvious that each individual architecture has its own strenghts and weaknesses by which a choice can be made. Applications that need to be highly reactive would benefit from Flux for example, and apps in which a lot of similar information is displayed throughout would be served well by an MVVM structure. Or perhaps when working in a larger team MVP would be the superior choice since it would allow for very fine-grained control of every individual layout.

Adding to this complexity are the respective adoption rates: MVP is becoming established as the go-to methodology for developing Android applications. This monoculture can be problematic when MVP is simply not a good fit.

Besides all this, the case still remains that *any* choice is better than ad-hoc, spaghetti code. Perhaps the most difficult part of these patterns are not their implementation but encouraging and convincing others of their usefulness. Hopefully this thesis will help with that.

However, that is not to say this form of programming doesn't have its rightful place: unstructured code is excellent for quickly making prototypes which need to serve as a proof of concept.

Profiling also proved essential to figuring out performance problems. It would be nearly impossible to find the source of bad performance when dealing with even moderately complex code without the profiling tools (a perfect example would be the data binding library's issues with instantiation).

It is also necessary to remember that architectural design patterns are not a magic bullet. And neither are they a one-size fits all solution to every app's specific problem domain. Dogmatically sticking to a certain methodology could turn out to be extremely harmful in the long term. Sometimes the overhead is just not worth it.

Personal Conclusion

This was a very worthwhile project for me to finish because I had always been interested in how to deeply reason about the structure of code. It is my sincere hope that this thesis will not only serve as my final accomplishment at HoWest but a reference which others may consult every so often.

Although this thesis was not directly coupled to my internship (coupling is an antipattern, anyway...) I was still able to use this research into my daily tasks. This is because these pattern transcend specific programming environments and their knowledge is useful in a great variety of situations. This universality made it very rewarding to translate it directly into a specific framework.

The fact that Android is reasonably unexplored in this area was also a big motivator for me because my contribution will perhaps be able to shape how Android Development evolves in a small way.

Even if this turns out to be wishful thinking, I personally still learned a great deal about the internals of Android and computational models on mobile devices in general. Some tools which I had previously not considered, such as the memory tracer, have unexpectedly become personal favorites.

My one regret about this thesis is that due to time contraints I wasn't able to test these patterns in a more extensive case study.

However in conclusion I still believe that a lot of problem areas remain unexplored, such as how to handle fragmentation and the differences in performance it brings along.

References

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

Advocating Against Android Fragments. (n.d.). Retrieved March 20, 2016, from https://corner.squareup.com/2014/10/advocating-against-android-fragments.html

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

C. Martin, R. (n.d.). Confreaks TV Keynote: Architecture the Lost Years - Ruby Midwest 2011. Retrieved March 20, 2016, from http://confreaks.tv/videos/rubymidwest2011-keynote-architecture-the-lost-years

Dagger ‡ A fast dependency injector for Android and Java. (n.d.). Retrieved March 20, 2016, from http://google.github.io/dagger/

Data Binding Guide Android Developers. (n.d.). Retrieved March 20, 2016, from https://developer.android.com/intl/es/tools/data-binding/guide.html

Design Patterns and Refactoring. (n.d.). Retrieved March 20, 2016, from/

Dupree, K. M. (n.d.). RxJava for Android App Development - O'Reilly Media. Retrieved March 20, 2016, from http://www.oreilly.com/programming/free/rxjava-for-android-app-development.csp

Evans, J. (n.d.). Tcpdump is amazing - Julia Evans. Retrieved March 20, 2016, from http://jvns.ca/blog/2016/03/16/tcpdump-is-amazing/

Facebook: MVC Does Not Scale, Use Flux Instead [Updated]. (n.d.).*InfoQ*. Retrieved March 20, 2016, from http://www.infoq.com/news/2014/05/facebook-mvc-flux#anch110016

Ferreira, D., Dey, A. K., & Kostakos, V. (2011). Understanding Human-Smartphone Concerns: A Study of Battery Life. In K. Lyons, J. Hightower, & E. M. Huang (Eds.), *Pervasive Computing*, Lecture Notes in Computer Science (pp. 19–33). Springer Berlin Heidelberg. Retrieved March 20, 2016, from http://link.springer.com/chapter/10.1007/978-3-642-21726-5_2

Flux Application Architecture for Building User Interfaces. (n.d.). Retrieved March 20, 2016, from http://facebook.github.io/flux/index.html

Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Booch, G. (1994).*Design Patterns: Elements of Reusable Object-Oriented Software* (1 edition.). Reading, Mass: Addison-Wesley Professional.

Google Developers. (2014, August). DAGGER 2 - A New Type of dependency injection. Retrieved March 20, 2016, from https://www.youtube.com/watch?v=oK_XtfXPkqw

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

Marshmallow Brings Data Bindings to Android, with Yigit Boyar and George Mount. (n.d.). Retrieved March 20, 2016, from https://realm.io/news/data-binding-android-boyar-mount/

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

ReactiveX - Operators. (n.d.). Retrieved March 20, 2016, from http://reactivex.io/documentation/operators.html#alphabetical

Richards, M. (n.d.). Software architecture patterns. *O'Reilly Media*. Retrieved March 20, 2016, from https://www.oreilly.com/ideas/software-architecture-patterns

Robolectric. (n.d.). Retrieved March 20, 2016, fromhttp://robolectric.org/

Robolectric vs Android Test Framework - Stack Overflow. (n.d.). Retrieved March 20, 2016, from http://stackoverflow.com/questions/18271474/robolectric-vs-android-test-framework

Sandin, M. (2016, February). Four Strategies for Organizing Code: The Whys and Whats of Organizing Code, Strategy #1—by Component, Strategy #2—by Toolbox, Strategy #3—by Layer, Strategy #4—by Kind, Summary. *Medium*. Retrieved March 20, 2016, fromhttps://medium.com/@msandin/strategies-for-organizing-code-2c9d690b6f33#.1jnehhnec

Square/mortar. (n.d.). *GitHub*. Retrieved March 20, 2016, fromhttps://github.com/square/mortar

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

The MVVM Pattern. (n.d.). Retrieved March 20, 2016, fromhttps://msdn.microsoft.com/en-us/library/hh848246.aspx

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/

What is Extensibility? (n.d.). Retrieved February 24, 2016a, fromhttps://msdn.microsoft.com/en-us/library/aa733737(v=vs.60).aspx

What is Extensibility? (n.d.). Retrieved March 20, 2016b, fromhttps://msdn.microsoft.com/en-us/library/aa733737(v=vs.60).aspx

Why is verbosity bad for a programming language? - Programmers Stack Exchange. (n.d.). Retrieved March 20, 2016, from http://programmers.stackexchange.com/questions/141175/why-is-verbosity-bad-for-a-programming-language