Contents

Licence

Figure 1:

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

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 2016, 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 : The Philosophy of Extensible Software" 2016)

```
 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 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.

```
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

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

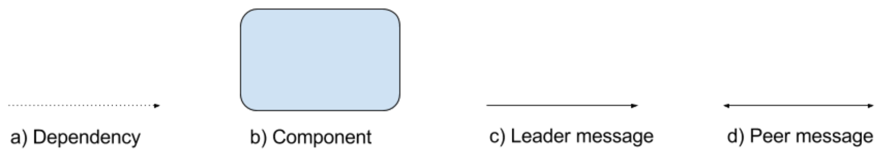

Figure 2: Conventions

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" 2016,)

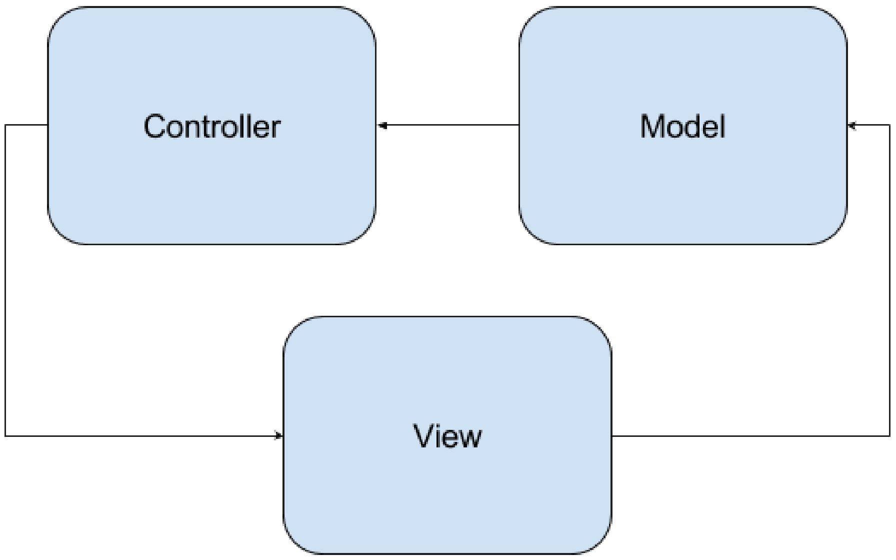

Figure 3: MVC

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.

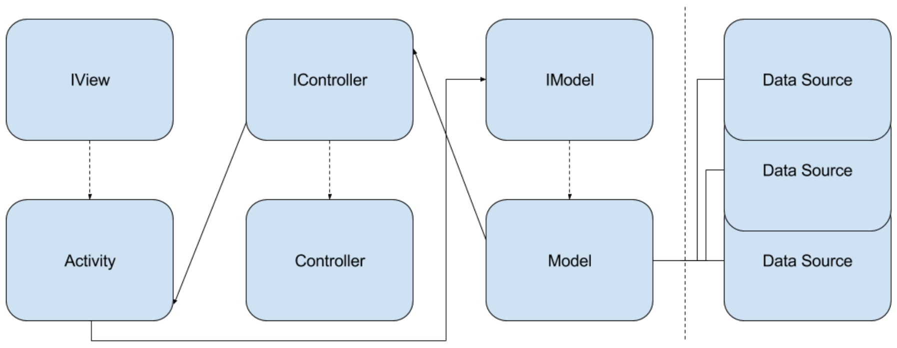

Figure 4: MVC Implementation

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: How to Organize the Presentation Layer. Antonio Leiva" 2014–2014-04-15T15:19:55+00:00)

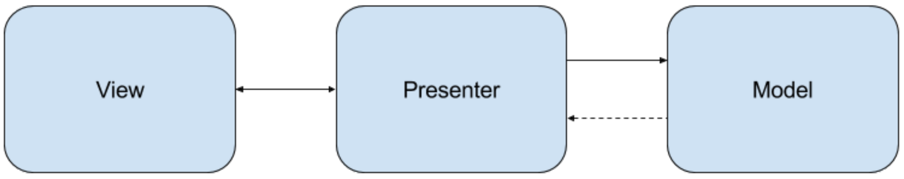

Figure 5: MVP

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.

MVVM

Model, viewmodel, view.

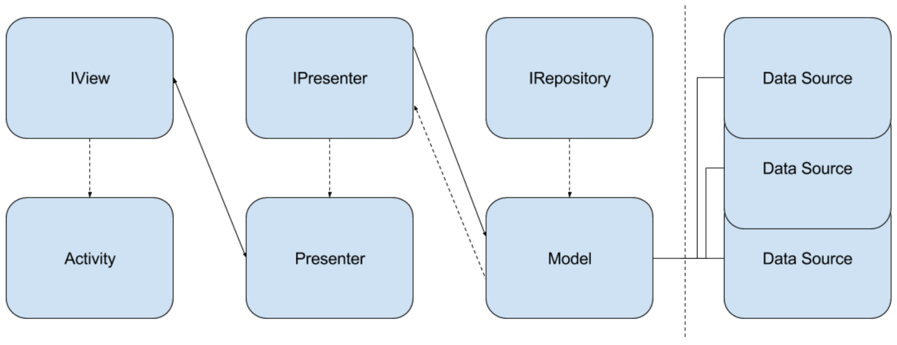

Figure 6: MVP Implementation

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.

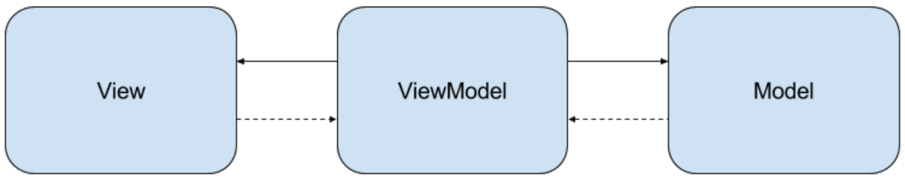

Figure 7: MVVM

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.

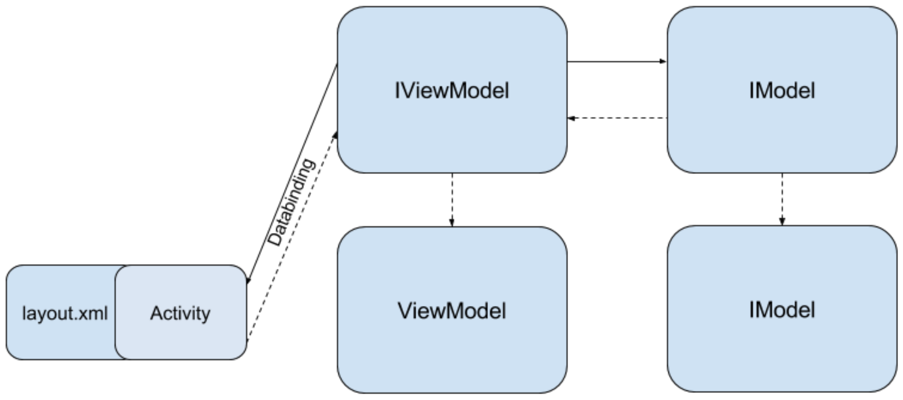

Figure 8: MVVM Implementation

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.

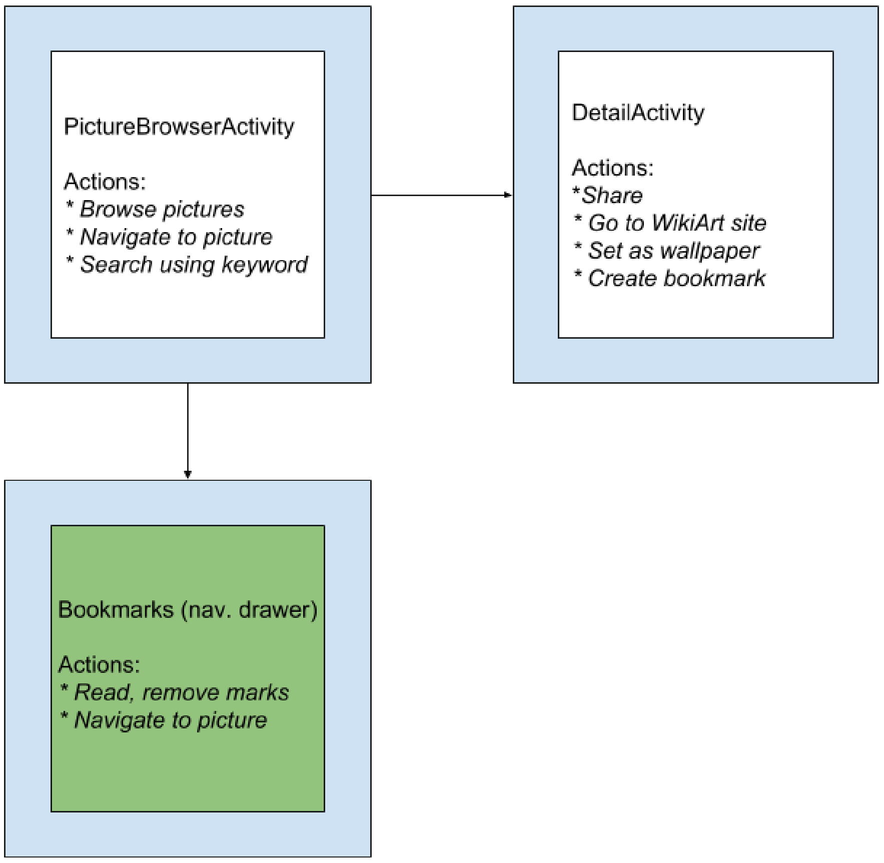

Figure 9: Application flow

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. GitHub" 2016) ("There Is a Special Place for Samsung in Android Hell - Anas Ambri" 2016).

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.

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:

Or would it take quite some effort:

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.

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.

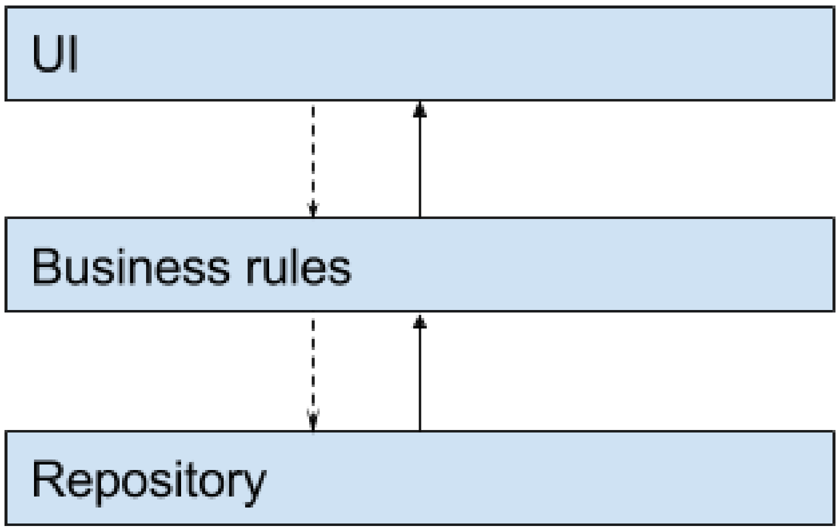

Figure 10: Decoupled code

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.

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.

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.

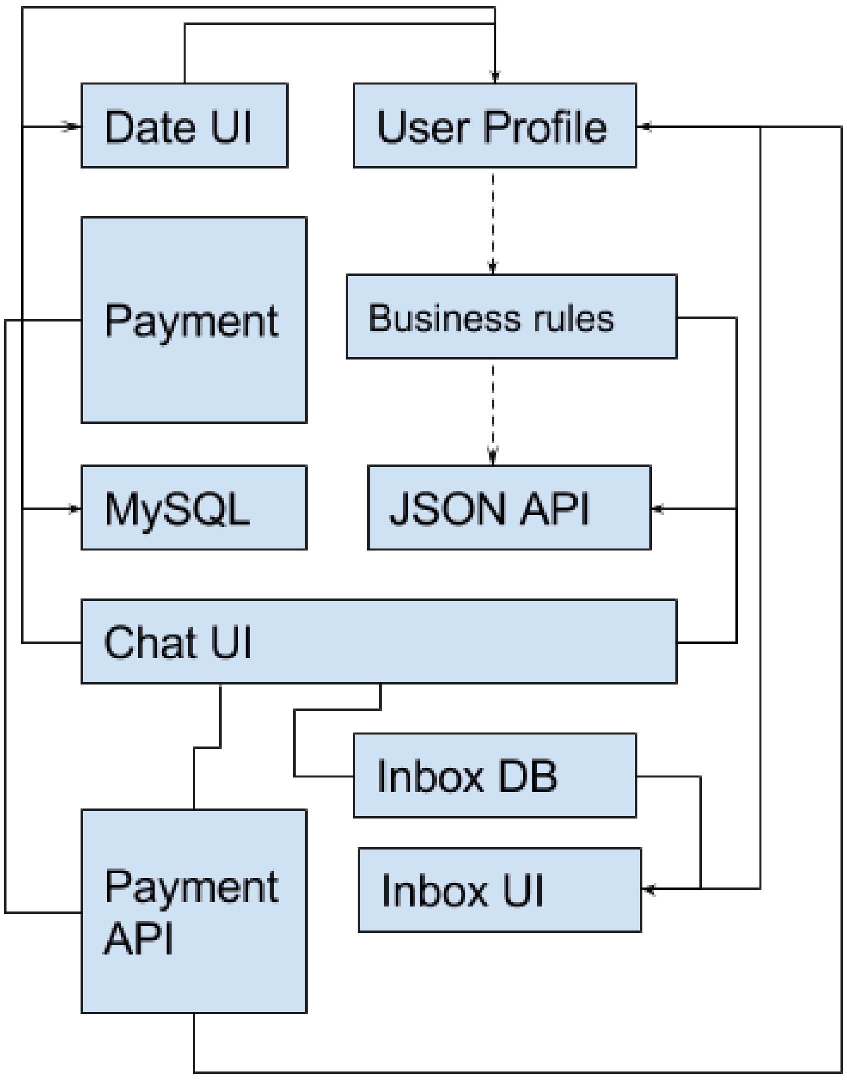

Figure 11: Entangled code

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.

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?" 2016) 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*.

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:

- 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).

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.

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.

Memory Performance

Testing Battery Usage 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: 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. This does not mean Battery Historian is difficult to work with however, as it only requires a few steps:

First, download Battery Historian from https://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
```

Bandwidth Efficiency Testing

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*.

Espresso

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.

This thesis's source code will be 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.

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 Rocks!");
    }
}
```

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).

JUnit JUnit is the standard testing framework for Java.

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.

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`

(**???**) 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 (**???**) 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).

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.

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<F
//The network call needed to receive a weather forecast is placed in a Callable, which mak

    @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 separate thread, so
    .observeOn(AndroidSchedulers.mainThread()) //The Observer is set to receive events on
    .subscribe(new Observer<List<Forecast>>() {

        @Override
        public void onCompleted() { } //This event is fired when the Observable is done em

        @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 th
            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.

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

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." 2016. Accessed February 24. http://accu.org/index.php/journals/391.

Beizer, Boris. 2003. *Software Testing Techniques.* Dreamtech.

"Icechen1/androidcompat. GitHub." 2016. Accessed February 24. https://github.

com/icechen1/androidcompat.

“MVP for Android: How to Organize the Presentation Layer. Antonio Leiva.” 2014–2014-04-15T15:19:55+00:00. http://antonioleiva.com/mvp-android/.

Nystrom, Bob. 2016. “Design Patterns Revisited · Game Programming Patterns. Game Programming Patterns.” Accessed February 21. http://gameprogrammingpatterns.com/design-patterns-revisited.html.

“Techniques for Fault Tolerance in Software.” 2016. Accessed February 24. http://srel.ee.duke.edu/sw_ft/node5.html.

“There Is a Special Place for Samsung in Android Hell - Anas Ambri.” 2016. Accessed February 24. http://verybadalloc.com/android/2015/12/19/special-place-for-samsung-in-android-hell/.

“What Is Extensibility?” 2016. Accessed February 24. https://msdn.microsoft.com/en-us/library/aa733737(v=vs.60).aspx.