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) the license.

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.

Introduction

Abstract

Acknowledgements

Glossary

Word	Definition
Design Pattern	sfsdfdf
MVP	sfsdfdf
MVC	sfsdfdf
Java	sfsdfdf
Android	sfsdfdf
Behavioral Pattern	sfsdfdf
Implementation	sfsdfdf
Asynchronous	sfsdfdf
Activity	sfsdfdf
Fragment	sfsdfdf

Basic Design Pattern Knowledge

Building Blocks

Even though non-architectural design patterns are not the focus here, it is still nessecary to know a few 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 considered such a basic nessecity that many languages will ship with it (such as Java's Obverser class(Nystrom, n.d., chap. II.6)).

Mediator

Overview of several common patterns

MVVM

Theory

Implementation in Android

MVP

Theory

Implementation in Android

MVC

Theory

Implementation in Android

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.

What's out of scope

Quality measurement

Performance

Performance will be measured using Android Studio's built-in profiler. 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 [Link naar samsung bugs].

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.

Development speed

This is more difficult to measure but certainly useful.

Verbosity of 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. If a certain design pattern look promising but proves difficult to implement it simply might not be worth the extra effort, since the 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.

References

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