System design document for Coffee Break

Version: 0.1

Date: torsdag den 20 april 2017

Author: Felix

This version overrides all previous versions.

# 1 Introduction

General info. What is this? What does it describe?

## 1.1 Definitions, acronyms and abbreviation

Todo-list – A list of different tasks that the user wishes to get done.

Task – The simple item that the user can add to his/her Todo-list

List task – A more complex version of a task, which creates the Task with the specified name and also contains a list of different tasks that the user can use to specify a list of tasks in a single place. This way, the user receives another option when they wish to organize/structure their Todo-list.

Time Category – The category which tasks can be sorted into which involve a certain timeframe

Label Category – Custom categories that the user can create through either adding custom labels/tags onto their task during creation, or through setting up static categories which are always visible.

Test-driven development – Abbreviated as TDD. Before any new code is written for the application, a test for the specified component will be make using the different specifications for the component as guidelines. This will lessen the number of bugs in the end product. The procedure can be read in depth at <https://blog.jetbrains.com/idea/2016/12/live-webinar-the-three-laws-of-tdd/> .

MVC – “Model, View, Controller”, a design model used in the most applications/programs today. The Model is the database which handles all of the logic and calculations. The View is the what is actually shown to the user, and the Controller handles the interaction between the user and the Model.

Object Oriented implementation – A certain form of programming paradigm, where the coding is divided into different objects and classes. This is to break up the different tasks into smaller, more manageable parts and then tackling the problem by creating one “puzzle piece” at a time.

# 2 System architecture

The most overall, top level, description of the system. Which (how many) machines are involved? What software on each (which versions). Which responsibility for each software? Dependency analysis. If more machines: How do they communicate? Describe the high level overall flow of some use case. How to start/stop system.

An ​UML deployment diagram​, possibly drawings and other explanations. Possibly UML sequence diagrams​ for flow.​

(Persistence and Access control further down)

Any general principles in application? Flow, creations, ...

# 3. Subsystem decomposition

For each identified software above (that we have implemented), describe it ...

## 3.1 “...First software to describe” ...

Recap: What is this doing (more detailed)

Divide it into top level subsystems. An ​UML package diagram for the top level. Describe responsibilities for each package (subsystem). Describe interface. Describe the flow of some use case inside this software. Try to identify abstraction layers. Dependency analysis Concurrency issues.

If a standalone application

* Here you describe how MVC is implemented
* Here you describe your design model (which should be in one package and build on the domain model)
* A class diagram for the design model.



Figure 1 MVC Design Pattern

### MVC, MVP and MVVM

The system of this application has been implemented using the standard MVC architectural design pattern. In an Android application, all MVC-based architectural flows will look similar to this depending on the specific implementation you choose.

For the Android API, there are three fitting MVC patterns: MVC, MVP and MVVM. These three implementations all share the same principle of separating the calculations and data into the Model and displaying the data independently inside the View. What distinguishes them from each other is the way that the Model and View communicate: through a Controller, Presenter or Viewmodel.

This application is based on the MVC principle itself, where the Controller is the brains of the operation. The Controller consists of all the implementation logic that the View uses to display the data of the Model, and the logic for interaction between the Model and the User. Data that will be displayed for the User is fetched from Model by the active Activity in the Controller, and then injected into the necessary XML-Layout to be viewed on the screen as the View. A view never communicates directly with the Model, because all the representational logic is implemented in the Controller. When the User interacts with the View, the Controller will be notified and tell the Model to make the necessary calculations. The results are then sent back to the Controller, which then updates the View.

#### Pros and Cons of MVC

For smaller applications such as this one, going with MVC instead of MVP and MVVM may be beneficial. More specifically, applications with few Activities and XML-Layouts that can be coupled together and don’t have many hierarchies won’t necessarily need to have another Controller to delegate the work between them. The Activities and Fragments themselves are enough for handling the communicational logic and don’t get too bloated in the process. The benefits will be less code to implement, and therefore less time needed for the same result.

The backside of the use of MVC instead of the other patterns is that the Controller and the View become very tightly coupled. In return, this makes the Controller hard to test using *TDD* due to the dependencies of the View, but also quite hard to *Maintain* as the Controller will easily get code added in the future, making it bloated and brittle. Using MVC also has a *Modularity and Flexibility* -issue. Due to the Controller being so tightly coupled with the View, it may even be an extension of the View itself. If the View will need to be changed in the future, then the Controller also will have to change. The better way of handling this issue would be to implement the application using either MVP or MVVM, where the Views consist of both the XML-layouts and their respective Activities/Fragments, and the Controller is an external class which only handles the information that the View will need to display.

### Diagrams

* Dependencies (​STAN​ or similar) - UML sequence diagrams for flow.

### Quality

* List of tests (or description where to find the test)
* Quality tool reports, like ​PMD​ (known issues listed here)

NOTE: Each Java, XML, etc. file should have a header comment: Author, responsibility, used by.., uses ...

## 3.2 “...next software to describe” ...

As above….

# 4. Persistent data management

How does the application store data (handle resources, icons, images, audio, …). When? How? URLs, pathe’s, … data formats… naming..

# 5. Access control and security

Different roles using the application (admin, user, …)? How is this handled?

# 6. References

The three laws of Test Driven Development - <https://blog.jetbrains.com/idea/2016/12/live-webinar-the-three-laws-of-tdd/>