System design document for Coffee Break

Version: 0.1

Date: torsdag den 25 maj 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.

POJO – An abbreviation for *“Plain old Java object”*, which is often used to quickly tell the readers that there are no real external dependencies and the implementation of the specific class/object is simply just Java-code.

JSON - CoffeeBreak stores its data in JSON, or JavaScript Object Notation, format between application sessions. JSON is a compact, text based format which main purpose is to transmit data.

# 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.

### Design Model

The model-package contains a couple of utility-classes for conversion and sorting, but is also contains the model-Class which acts as the entry point for the core of the application. This core consists of the tododatamodule package. Inside this package the core has been separated into three packages to simplify the structure of the program. The ToDoDataModule class is a façade pursuant to the façade pattern, which means it acts as a simplified interface to the different parts of the core.

* CategoryList

The CategoryList package is responsible for the categories that are used to filter tasks. All of the categories that exists by default and the user created categories is stored in the categoryList class. The categories are separated into two different types, TimeCategory and LabelCategory. The two main differences are:

The time based categories are never given to a task like a LabelCategory is. Instead a TimeCategory is dependent on the date that can be given to a task.

By default it exists four categories (today, tomorrow, next 7 days, next 30 days), these are all the time based categories that can be created. LabelCategories on the other hand can be created by the user to organize his tasks.

The Filtration of tasks is handled by the getValidTasks() method that every category implements, it takes a list of tasks as arguments and returns a new list with only the tasks that fits the criteria of the category.

* Statistics

The main purpose of the Statistics package is to listen and keep a record of certain user interactions. When the user has done an action a specific number of times, the user will be rewarded with an achievement.//not finished

* TodoList

The TodoList package is the most important part of the application because it is responsible for the tasks. There are three types of tasks:

The Task class represents the most basic form of a task, it contains a name, a boolean to tell if it’s checked or unchecked and the time and date of its creation. Every other type of task is assigned one or more instances of this class upon ther creation.

The AdvancedTask is the main tasks that the user will interact with, beyond being assigned an instance of the Task class it can also be assigned a date, a priority, one or more LabelCategories, and a description/note.

The ListTask is very similar to the AdvancedTask with the exception that ListTask holds a list with instances of the Task class, beyond the one that is assigned upon creation.



Figure 1 MVP Design Pattern

### MVC, MVP and MVVM

The system of this application has been implemented using the standard MVC architectural design principle. In an Android application, all MVC-based architectural flows will look similar to this independently of 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 a Viewmodel.

This application is based on the MVP-pattern, where the Presenter acts as the brains of the operation. Instead of having a Controller which consists of all the implementation logic that the View needs to display the data of the Model, the Presenter only handles the communicative logic between the Model and the Views. More precisely, the Presenter keeps track of the user interaction in the View and modifies the Model accordingly, but the View itself handles the logic for what it should display. With that said, the necessary Data that will be displayed for the User is fetched from Model by the Presenter and handed to its associated Activity in the View. The Activity then inflates the necessary XML-Layout to show on the screen.

An Activity sort of acts as a small Controller for each specific View, and the Presenter is the communicator between the Activity and the Model, only telling information that is necessary to be shared between the two. This way, the View never communicates directly with the Model, due to all the representational logic being implemented in the Activity. When the User interacts with the View, the Activity will notify the Presenter, which then tells the Model to make the necessary calculations. Conversely, the results are then sent back to the Activity via the Presenter, which then updates the View.

#### Pros and Cons of MVP

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 a delegating Controller to handle the work between the rest. The Activities and Fragments themselves may be 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.

Therefore, we chose to go with MVP. Even though implementing the application in this fashion means more code, MVP handles these issues in a better manner. Having the Views consist of both the XML-layouts and their respective Activities/Fragments, and letting the Controller for each View be the Presenter in the form of an external class allows for better *Modularity*, *Flexibility* and *Reusability* of code. In return, this gives a more *Maintainable* end product.

### Diagrams

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

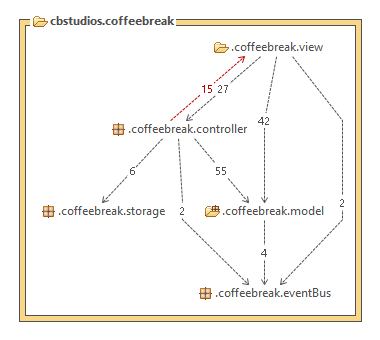
The STAN diagram above shows the dependencies between the four different main packages: Model, View, Controller and Storage/Util, plus the used Android Event Bus library.

Figure 2 STAN Diagram of the MVP structure

So far, there aren’t any circular dependencies, except for those between the Controllers and Views. However, this circular dependency is necessary due to the Presenter in the Controller package acts as the mediator between the View and Model. Therefore, it gets called by the Activity in the View when the User interacts with the application. The dependency in the reverse direction is due to the Presenter returning information from the Model when the View is to be updated in a major way. This is one of the trade-offs for choosing to go with MVP, because the “Controller” of MVC is split into two, the mediating Presenter and the structuring Activity. (See [MVC, MVP and MVVM](#_MVC,_MVP_and) for more information.)

The direct dependencies between the Model and the View packages is due to the Adapter class and its inherent ViewHolder class, which represent each existing task in the categorized list and updating their respective Task directly when its state is changed.

#### EventBus implementation

Event handling and communication between components is handled through the external library *“EventBus”* by Greenrobot. This library implements the publisher/subscriber pattern to achieve loose coupling, and does so in a very simple and efficient fashion. All the different pros and features can be found on <http://greenrobot.org/eventbus/>, but the reason this library was chosen is the fact that only three steps are needed to implement a functioning event with Subscribers and Publishers.

Firstly, each specific event is implemented through POJO, without any specific requirements.

When it’s then time to prepare Subscribers, they implement *“event handling methods”* (<http://greenrobot.org/eventbus/documentation/how-to-get-started/>) and are then annotated with the @Subscribe annotation. These methods have no naming conventions as of the latest major release of EventBus 3.

For the actual subscriptions, the Subscribing classes have to register and unregister themselves from the bus, as to control when the events will be acted upon.

Finally, for posting the actual events you only need to call the EventBus and post a specific event-object on the bus. Each subscribing class will then act on the posted event.

This simple way of handling events is very useful in many aspects. E.g. when it comes to passing data between Activities. It isn’t possible to pass specific objects between Activities directly because they can’t be created and run through direct constructors in the Android OS. The more complex way of handling this issue would be to pass the necessary data in String-format in the Activity’s Intent-object and then have the new Activity fetch the corresponding data from the Model. This would theoretically lead to the same result, but take considerably more time to implement and more resources during Runtime.

#### Sequence Diagrams for use cases

The first and most important use case is to create a new task. As shown in figure 3 below it starts with the user taps on the floating action button and chooses “Advanced Task”. The method addAdvTask in MainActivity will be called and get access to the ToDoDataModue via the Model. The createTask method in the ToDoDataModule will be called and go through the ToDoList and on to the TaskFactory and create a new AdvancedTask. The new AdvancedTask will be placed in a list in the ToDoList class.

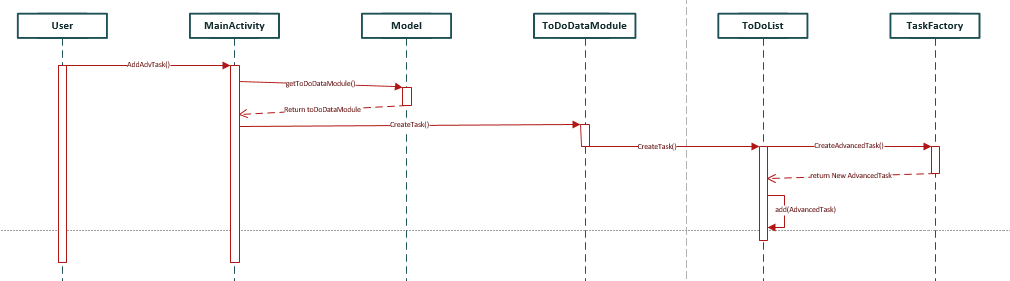


Figure 3 Sequence diagram of the use case -Create Task

The second use case is to check off a task. It is done by the user taps the check off box on the screen. The TasksAdapter receives the tap and calls the setChecked method in the AdvancedTask class which then calls setChecked in the Task class as seen in figure 4 below. The setChecked method in Task will change the Task Boolean to true if the task was in fact checked off. It will do the opposite if the user clicks on the check off box when it is already checked.

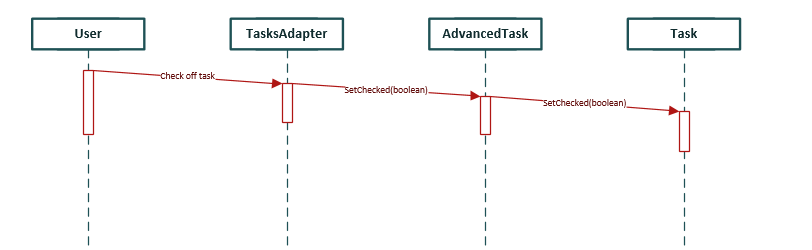


Figure 4 Sequence diagram of the use case -Check off task

### Quality

All tests are found in *Application/CoffeeBreak/app/src/test/java/cbstudios/coffeebreak/*. The name of each test is a description of what the test tests.

Some of the tests are specifically for use-cases. These test the functionality of the software on a higher level compared to the other tests. The name of such a test is marked with *UC*.

#### List of tests

All sub items in the list are the Use Cases that each test handles:

* CategoryListTest
* ModelTest
* MultipleDayTimeCategoryTest
* SaveAndLoadTest
* SingleDayTimeCategoryTest
* TaskEqualsAndHashcodeTest
* TaskSorterTest
  + “Sort task in another order:”
    - Priority level
    - Chronological order
    - Alphabetical order
* ToDoDataModuleTest
  + “Delete task” – testRemoveTask()
  + “Create a new task in the form of:” – Both in testAddTask()
    - General Task
    - List Task
  + “Update/clean up list of done tasks” – testUpdateToDoList()
  + “Create new Label category” – testAddLabelCategory()
  + “Delete a Label category” – testRemoveLabelCategory()
  + “Filter the list to a certain category” – testFilterTasksByCategory()
* UCCreateAndCheckTaskTest
  + Check off a task
* UCChangeTaskConfiguration
  + Update a task’s configuration
* 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

Due to JSON being text-based, it isn’t the most optimal way to store data. The most optimal way would be to use a proper database to store the data, i.e. MySQL. Despite this, JSON was chosen due to the extra resources implementing a proper database would need. Instead, these resources could be used on more critical parts of the application.

Data storage in CoffeeBreak consists of two main components, the class StorageUtil and a data converter. StorageUtil is the class responsible for saving and loading the data to/from file. The data converter is responsible for converting data between JSON format and java objects.

StorageUtil is designed to work in an Android-environment. The perform a save/load, the class needs an Android Context and a String identifier. The context is used to determine where on the android device it should save the data and the identifier is used to be able to save/load different sets of data. The identifier is used as the filename for the data when saving/loading it.

The data converters are responsible for converting data. They convert objects into JSON and JSON back into these objects. Because of how different classes may be structured, there needs to be specific converters for each class.

To handle the conversion and parsing of JSON, Google’s JSON library Gson is used. The main reasoning for using Gson instead of the native implementation of JSON in Android is the fact that Android implementation requires an Android runtime environment to function. This makes it impossible to do tests for the converters, whereas Gson doesn’t have this limitation.

Data is stored when the Android system calls the *onPause()*-method in the application. This method is called when the application loses focus in any way. This ensures data always gets saved before exit of the application.

# 5. Access control and security

CoffeeBreak doesn’t use roles, as it’s designed to be used by a single individual. This essentially makes this single user administrator. This type of application doesn’t need many administrator tools and it’s designed to be user-friendly and easy to use as well. This makes most of the admin-tools obsolete.

For the few tools that the user needs, a simple settings interface is enough.

# 6. References

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

Greenrobot EventBus-library for Android - <http://greenrobot.org/eventbus/>

Greenrobot EventBus, how to get started - <http://greenrobot.org/eventbus/documentation/how-to-get-started/>

JavaScript Object Notation - <https://en.wikipedia.org/wiki/JSON>

Android Internal Storage - <https://developer.android.com/guide/topics/data/data-storage.html#filesInternal>