SYSTEM DESIGN DOCUMENT FOR COFFEE BREAK

VERSION: 2.0

DATE: SÖNDAG DEN 28 MAJ 2017

AUTHOR: FELIX, ZACK, ELIAS AND JOHAN

This version overrides all previous versions.

1 INTRODUCTION

This Software Design Document provides documentation which will give a clear picture of how this software is developed and designed. It will give guidance to developers with graphical and narrative documentation. This includes design models, use case models, sequence diagrams, and other information which describes the software and how it works.

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 what is actually shown to the user, and the Controller handles the interaction between the user and the Model.

MVP – “Model, View, Presenter”, an alternate form of the MVC-model. Here, the Controllers’ responsibilities are being split in two separate classes. One that handles the View-management logic, which is placed together with the View in the view package. The other handles communication between the Model and the View, which is in the form of a Presenter. See https://news.realm.io/news/eric-maxwell-mvc-mvp-and-mvvm-on-android/ for a more in-depth description of the differences between MVC and MVP.

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 – Coffee Break 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.

(Persistence and Access control further down)

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

Coffee Break is a standalone application, which runs locally on the user's mobile Android device. The only outward dependency that the application has is it's need to store persistent data on the device's storage between runs. More information about this can be found in section 4 of this document.

DEPLOYMENT PROCESS OF THE APPLICATION

When the application is executed by the user, its MainActivity-class in the view package is being instatiated first by the definition in the application's manifest. When this object then runs its onCreate()-method, it calls for the PresenterFactory to instatiate the DelegatingPresenter-class.

The DelegatingPresenter-object then loads in all persistent data from the device, creates a presenter for the MainActivity-object and then takes on the responsibility of managing all other presenter requests from activities when they are launched.

The in-depth process is a bit more complex than described in the previous paragraphs for the initialization of the application. See Figure 1 below for a basic deployment diagram of the 13-step process.

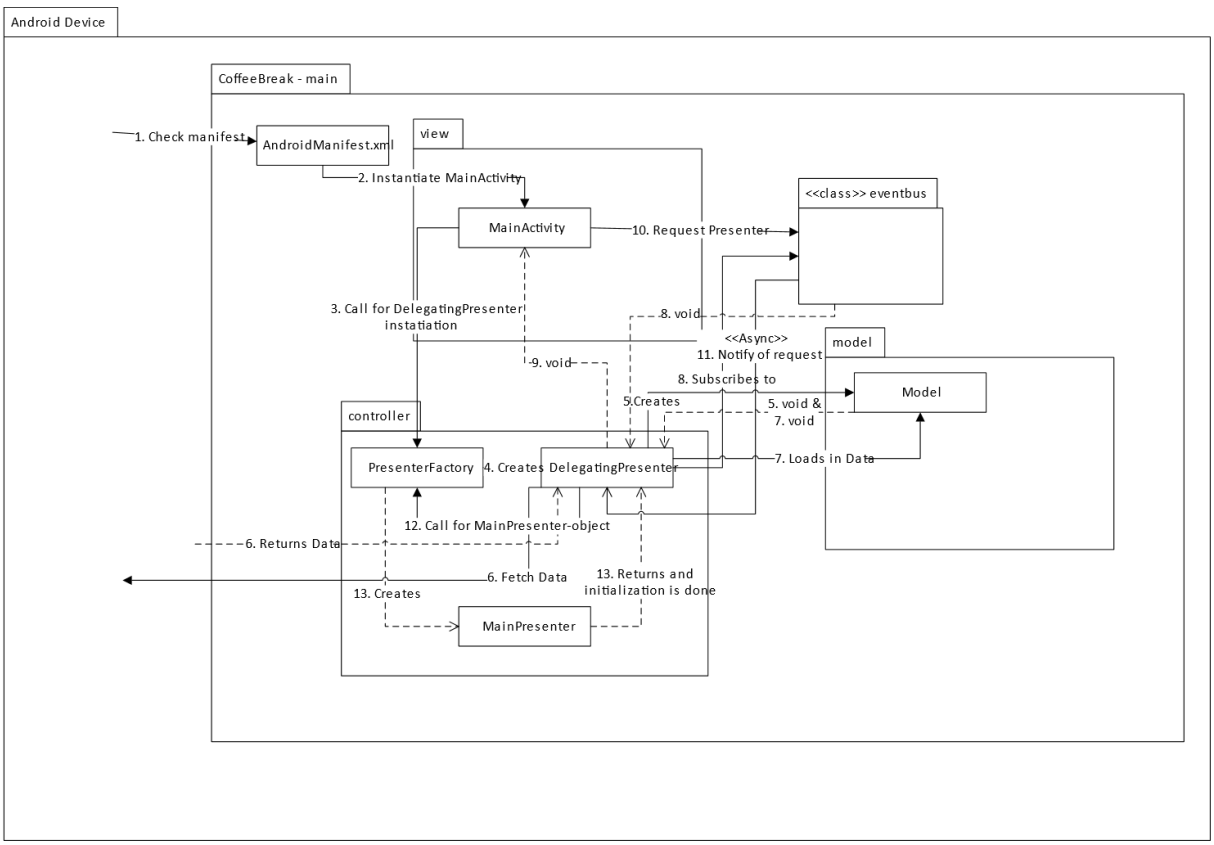

FIGURE 1 DEPLOYMENT DIAGRAM FOR THE INITIALIZATION

3. SUBSYSTEM DECOMPOSITION

MVC, MVP AND MVVM

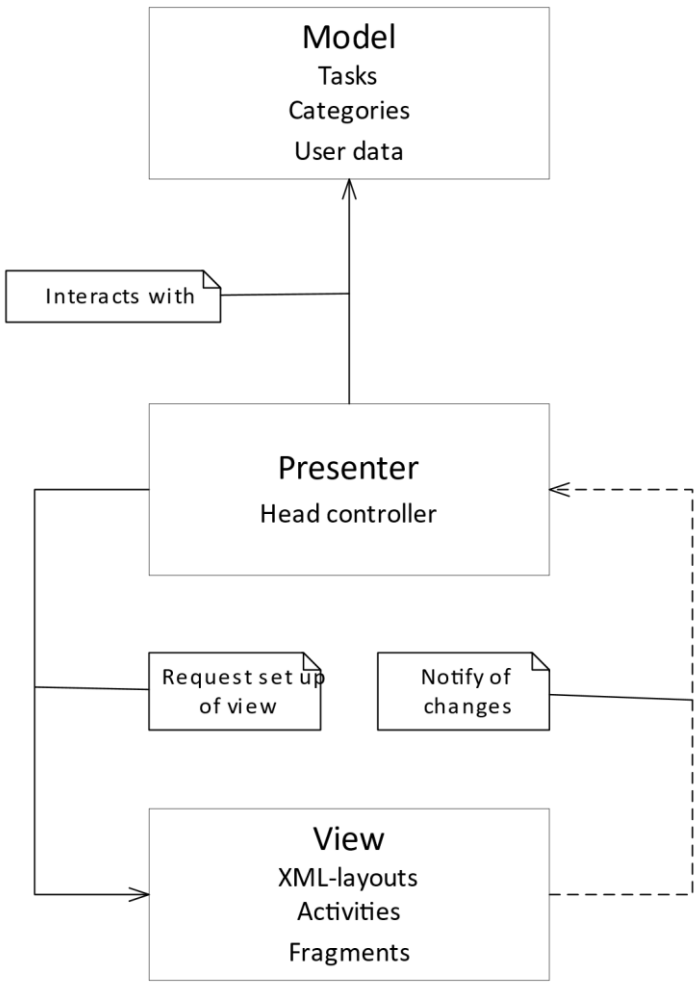

FIGURE 2 MVP-ARCHITECURAL PATTERN IN ANDROID

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.

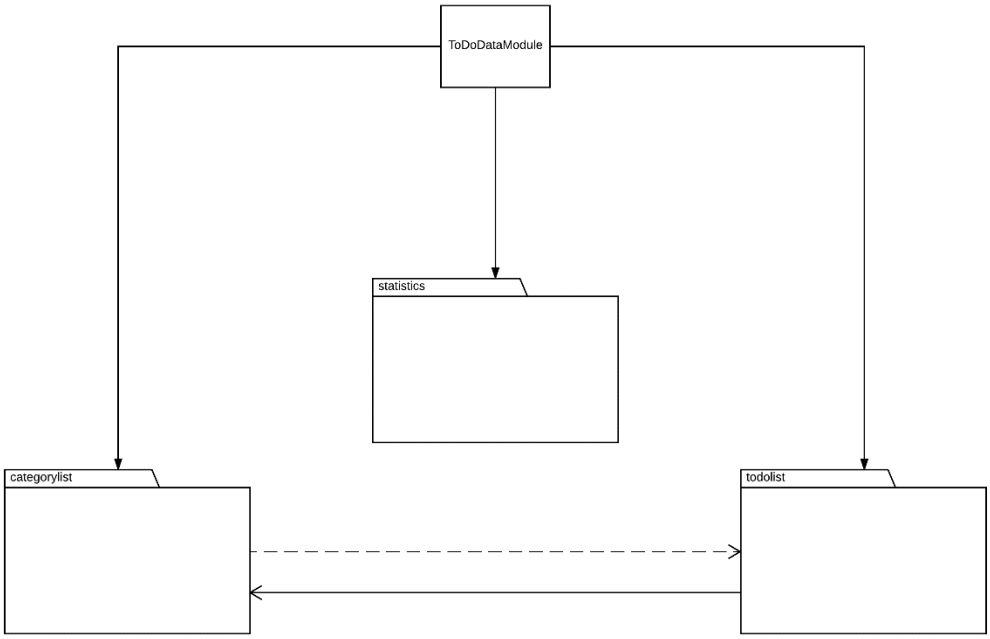

FIGURE 3 PACKAGING STRUCTURE OF THE MODEL

DESIGN MODEL

The model-package contains a couple of utility-classes for conversion and sorting. However, primarily it contains the model-Class which acts as the entry point for the application's core. The core consists of the "tododatamodule"-package, where the core has been separated into three smaller 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

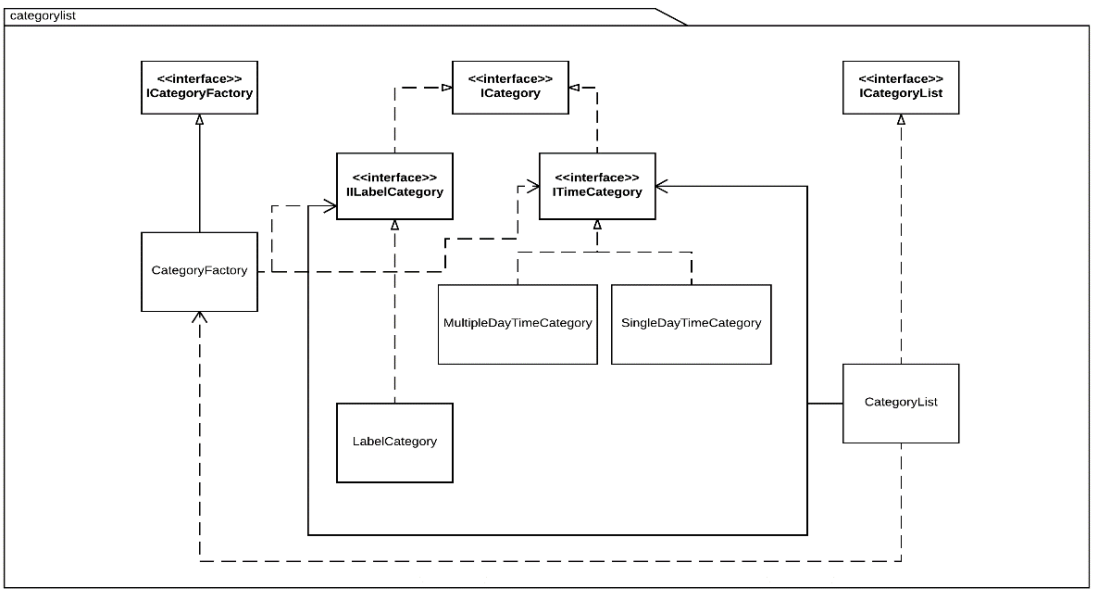

FIGURE 4 DETAILED VIEW OF THE CATEGORYLIST-PACKAGE

The CategoryList package is responsible for the categories that are used for filtering tasks. All of the categories that exist by default and the user created categories are stored in the CategoryList class. The categories are separated into two different types, TimeCategory and LabelCategory, where their 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, there are four time-based 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 the created tasks.

The filtration of tasks is handled by the getValidTasks()-method that every Category-class 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

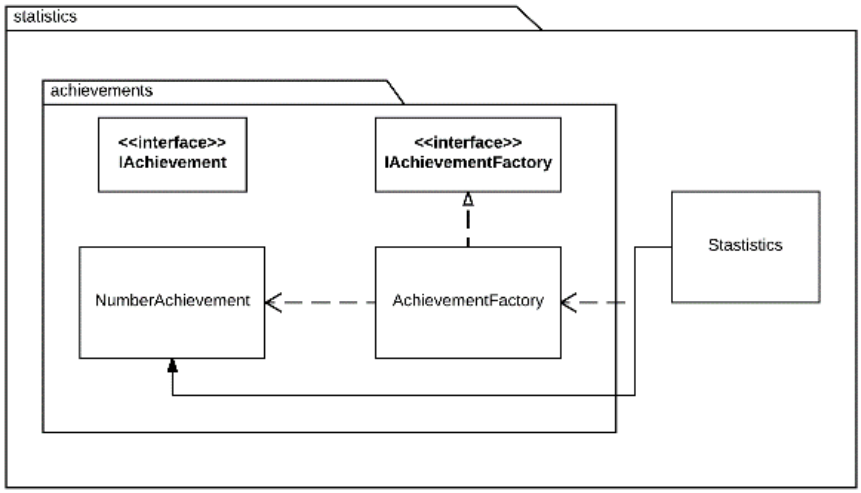

FIGURE 5 DETAILED VIEW OF THE STATISTICS-PACKAGE

The main purpose of the Statistics package is to keep a record of user interactions. When certain actions are performed different parts of the project will post events to the EventBus. These will be caught in the Presenters whom will update the appropriate fields in the Statistics class. When the user has done an action a specific number of times, the user will be rewarded with an achievement.

TODOLIST

The TodoList package is the most important part of the application. It is responsible for handling the tasks.

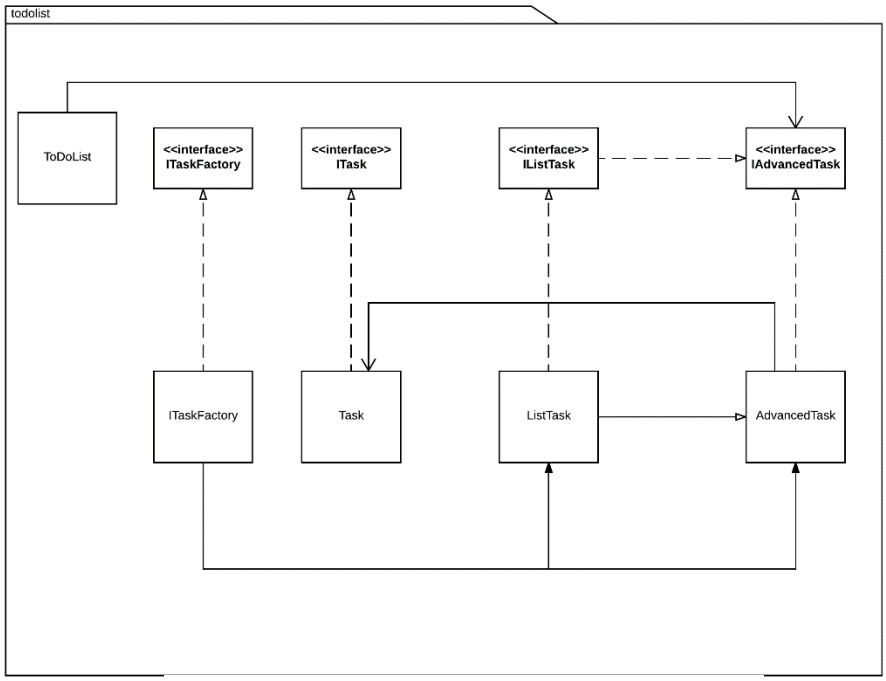

FIGURE 6 DETAILED VIEW OF THE TODOLIST-PACKAGE

There are three types of tasks, which are described below.

TASK

The Task class represents the most basic form of a task. It contains a name, a boolean value to tell if it's checked or unchecked and the time and date of its creation. Every other type of task is assigned an instance of this class upon their creation.

ADVANCEDTASK

The AdvancedTask is the primary form of task that the user will utilize. Aside from 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.

LISTTASK

The ListTask is very similar to AdvancedTask, with the exception that ListTask holds a list of instances of the Task-class, beyond the one that is assigned upon creation. The ListTask is fully implemented except that it does not have a visual representation of the subtasks, nor is it possible to interact or create subtasks.

INTERFACES

Interfaces have been heavily used throughout the whole application, especially in the model where almost every concrete class is implementing an interface. The reason for this is to make the code modular, exstensible and easy to maintain. Because when you are dependent on interfaces instead of concrete classes, you have the ability to swap the concrete implementiation to a new or modified one as long as it implements the same interface.

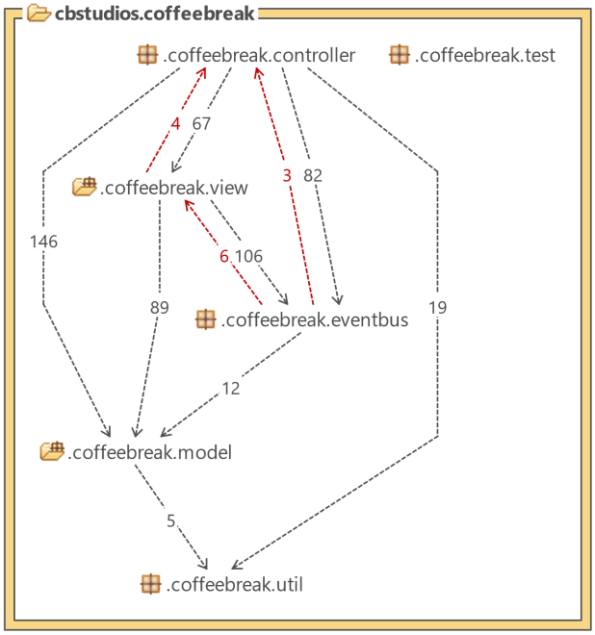

FIGURE 7 STAN DEPENDENCY DIAGRAM OF THE MVC-PACKAGES

DIAGRAMS

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.

Except for dependencies outside the event bus, there aren't any unnecessary circular ones. The four from the view-package to controller-package, however, are necessary dependencies due to the DelegatingPresenter in the Controller package needs to be initialized and instantiated. Since our MainActivity in the view-package is the first object being instantiated when the application runs, it then needs to create the DelegatingPresenter through the PresenterFactory.

Afterwards, the DelegatingPresenter handles all further presenter instatiation, dependency injection and model instatiation. This is one of the trade-offs for choosing to go with MVP in this sort of fashion, because the "Controller" of MVC is split into two, the mediating Presenter and the structuring Activity. (See MVC, MVP and MVVM for more information.)

The direct dependencies between the Model and the View packages is due to the Activites' data dependencies in their different components, such as the Adapter class and its inherent ViewHolder class. These classes represent each existing task in the categorized list and updates their represented 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 must 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 8 below, it starts with the user tapping on the floating action button and chooses their preferred task. The method addAdvTask()/addListTask() in MainActivity will be called and the activity then posts a creation request to the event bus, which gets handled by MainPresenter.

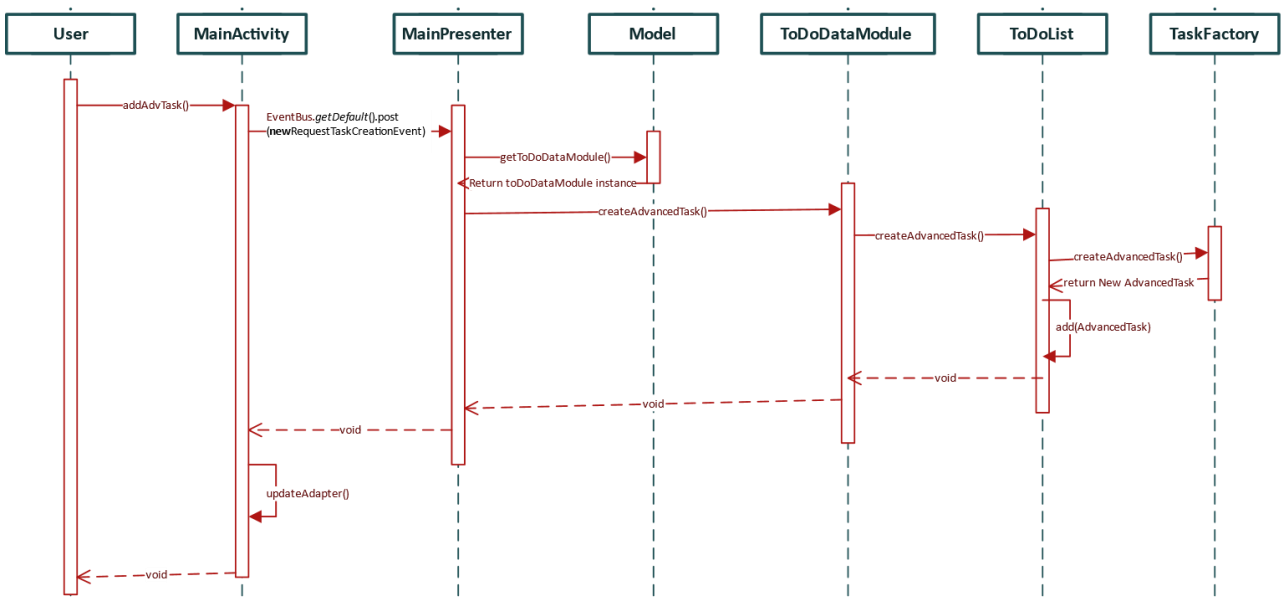

FIGURE 8 SEQUENCE DIAGRAM OF USE CASE "CREATE TASK"

The presenter then accesses the ToDoDataModue via the Model and calls the createTask()-method. When this method is called, a cascading chain of calls happen down to the ToDoList-object, which calls the TaskFactory to create a new AdvancedTask or ListTask. The created task is then placed in the list of tasks in the ToDoList-class.

The second use case is to check off a task. This happens when the user first taps the check box on a task. The TaskAdapter receives a notification of interaction and the ViewHolder that represents the task in the adapter calls the setChecked()-method in the represented Advanced-/ListTask-instance, which then calls setChecked() in its Task-object, as seen in

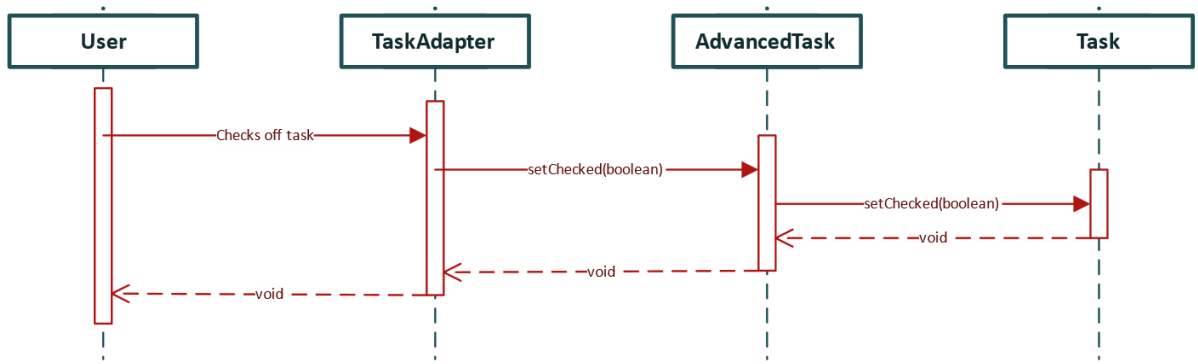

FIGURE 9 SEQUENCE DIAGRAM OF USE CASE "CHECK OFF TASK"

Figure 9 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.

QUALITY

Project:
CoffeeBreak
Scope:
Module 'app'
Profile:
Default
Results:
Enabled coding rules: 86

- PMD: 86

  Problems found: 0

- PMD: 0

Time statistics:

- Started: Sun May 28 17:48:26 CEST 2017
  - Initialization: 00:00:00.019
  - PMD: 00:00:00.989
  - Gathering results: 00:00:00.001
- Finished: Sun May 28 17:48:27 CEST 2017

FIGURE 10 PMD-RESULTS

PMD-tests have been used to ensure that suboptimal or unclear code is kept to a minimum. After some refactoring, all problems found were resolved.

All tests are found in *Application/CoffeeBreak/app/src/test/java/cbstudios/coffeebreak/* and are implemented to ascertain the functionality of each component in the application. The name of each test is a description of what will be tested.

The list below shows all of the different tests. Their underlying items each test has implies what use case they test.

LIST OF TESTS

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

- CategoryListTest
- ModelTest
- MultipleDayTimeCategoryTest
- SaveAndLoadTest
- SingleDayTimeCategoryTest
- StatisticTest
  - ”Check Statistics”
- AchievementTest
  - ”Check Achievements”
- 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()
    - Through Task creation/update
    - Static category creation
  - “Delete a Label category” – testRemoveLabelCategory()
  - “Filter the list to a certain category” – testFilterTasksByCategory()
- UCCreateAndCheckTaskTest
  - “Check off a task”
- UCChangeTaskConfiguration
  - “Update a task’s configuration”

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, a string identifier and a JsonElement. 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. JsonElement is a class provided by Gson which represents any type of Json data.

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

Coffee Break doesn't use roles, as it's designed to be used by a single individual. Therefore, the only user is the administrator. Most of the common administrator tools becomes obsolete with this application because of its design, which aims to be user-friendly and easy to use.

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
MVC, MVP and MVVM Smackdown - https://news.realm.io/news/eric-maxwell-mvc-mvp-and-mvvm-on-android/