Peer review - Boardbook

Design principles

- *Do the design and implementation follow design principles?*
    - *Does the project use a consistent coding style?*
    - *Is the code reusable?*
    - *Is it easy to maintain?*
    - *Can we easily add/remove functionality?*
    - *Are design patterns used?*

Firstly, in the Entity folder we see that the OCP (Open-Closed principle) is implemented. AbstractEntity is the superclass and the remaining classes as subclasses. As the principle goes, we want it open for extension and closed for modification as much as possible. In this case having an abstract class game having everything a board game should have, then having subclasses for the supported games in the app would be much efficient for code reuse, less dependencies between the game named classes and of course making it open for extension and closed for modification.

Some of the classes in entity only contains a single field such as a string, list of things or a single id. However the abstraction is understandable and we see the point in using them. But having something more they have in common would be more optimal. For example Game, GameRole and GameTeam all contain the String called name.
The same goes for the match related classes. Having an abstract class.

The observer pattern can clearly be seen used within all the presenter classes as listeners that listens for incoming changes, and updates the UI.

A standard singleton is seen within the Boardbook singleton class which is basically used to the whole application only is instantiated once, the class doesn't fit so well in the presenter package though.

They also use Adapter pattern e.g in places like GameDetails, GameDetailsRoles and etc. They use the adapter as a bridge between the UI components and data source that helps them fill the data in UI components.

Documentation

- *Is the code documented?*

Both the SDD and RAD are pretty useless right now as they don't contain anything. There is also a lack of comments in the code making it hard to understand.

Names

- *Are proper names used?*

Naming of the classes feels good as they relate much to the domain model. They are easy to understand, analyze and work with.

Modularity and dependencies

- *Is the design modular? Are there any unnecessary dependencies?*

By looking at the UML there seems to be good modularity between the different layers. There are many interfaces that are provided with many of the classes so they can easily be swapped out if needed. There are almost no libraries as well which means that this can be easily moved somewhere else without libraries being part of an issue.
On a closer look the BoardBook seems to be used as kind of a “god” object that is responsible for handing out instances for many other classes. Maybe this is good, maybe this is bad. I don’t know.

Abstractions

- *Does the code use proper abstractions?*

There seems to be some code duplication, for example instead of using an anonymous class of the TextWatcher, it can be made into a normal class to avoid code duplication.

They use abstraction as in encapsulation, having their global variables private and only be accessible with getters and setters.

A good abstraction is also where GameListAdapter and GameGridAdapter both extend the GameAdapter class.

They’ve made a ChatGroupFactory, but haven’t implemented anything. If they implement the factory pattern, it’ll give abstraction to the constructor for creating group chats and giving less dependencies between classes.

Testing

- *Is the code well tested?*

We found only three tests so there seems to be a lack of it. Maybe test-driven development hasn't been followed? There seems to be a test, TestUtil, in the production folder structure? With such a big model some more tests would come in handy, and help greatly towards the discovery of bugs when the code is updated.

Security and performance

- *Are there any security problems, are there any performance issues?*

As the UI is very thin at this moment, the performance seems to be very good. Didn't get the search to function so couldn't test that.

Architecture

- *Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?*

There seems to be an MVP architecture with clearly named folder structure. The model is very isolated since it all is plan java code with no dependencies. However you could not take out the model in this instance since everything that uses the model depends on the model and not an interface that's in front of the model. This is easily implemented however.

Improvements

- *Can the design or code be improved? Are there better solutions?*

You seem to have built everything on old technology and a lot of boilerplate code is present. You should consider using new libraries to not reinvent the wheel.

There are a lot of classes in the repository section, half of them seems to only convert an existing class to a class being able to be saved on firebase or used easier with firebase?. We understand that this might be for modularity, but if you were to change to another database you would have to change both the repo and the repo converters. If the toMap function is what you need, it would probably be easier put the toMap in the respective entity's code, or create a single converter class that act differently depending on what it is converting.

There's a lot of "unnecessary" boilerplate code where new TextWatchers are created to listen to textChanged events. You could create a class that extends the TextWatcher that acts differently depending on the context it is instantiated, instead of creating new local TextWatchers all the time. Instead of letting the activity implement an interface for example IAccountManager.

Instead of having all the presenter classes, livedata could be used to easily update within the fragment itself and no presenter class would be necessary.

An annoyance as a user would be that the apps logs you out everytime you close it, it would feel a lot better if you stayed logged in.