COS30017 – Custom project final report (level 3)

Ta Quang Tung – 104222196

GitHub repo: https://github.com/SoftDevMobDevJan2024/customapp-104222196

Contents

Idea and vision

My custom app idea is to build a smart home app that allows users to conveniently manage their smart appliances. Users can add or remove devices and control them on their phones. Due to time constraints, my app will not be interacting with real devices, but rather virtual devices stored in the local database.

To fulfill this vision, I have defined several user stories and their corresponding use cases:

User story: As a smart home owner, I would like to see a list of all my devices in one place.	
User interaction	App response
The user clicks on the "Devices" tab on the bottom navigation bar.	The app takes the user to a page with a list of device categories (lights, air conditioners, etc.)
The user clicks on a category.	The app takes the user to a page with a list of devices belonging to that category.
The user clicks on a device in the list.	The app takes the user to a page containing the device's details such as its name, location, status, and other settings.

User story: As a smart home owner, I would like to keep my most frequently used devices in one place for quicker access.	
User interaction	App response
In the list of devices, the user clicks on a specific device.	The app takes the user to the device's detail page, which contains the option to add the device to the home page.
The user clicks on "Add device to homepage".	The app adds the device to the homepage and shows a status message informing the user that the device has been added.
The user clicks on the "Home" tab on the bottom navigation bar.	The app takes the user to the homepage, which lists all the devices they have added.

User story: As a smart home owner, I would like to add a new device to my home.	
User interaction	App response
The user clicks on the "Devices" tab at the bottom navigation bar.	The app takes the user to the Devices page, which contains a floating action button at the bottom right corner of the screen.
The user taps on the button.	The app takes the user to a page where they can specify the details of the device they want to add.
The user enters the device's details and taps "Add device".	The app adds the device to the user's home and redirects them to the device list where they can see the newly added device.

User story: As a smart home owner, I would like to adjust the settings of my devices in the app.	
User interaction	App response
In the list of devices, the user clicks on a specific device.	The app takes the user to a page containing the device's details and settings.

The user interacts with one or more UI widgets to control the device.	The app updates the device's settings and notifies the user of this change.

User story: As a smart home owner, I would like to be able to modify the details of my device such as its name or location.	
User interaction	App response
In the list of devices, the user clicks on a specific device.	The app takes the user to a page containing the device's details and settings.
The user taps on the "Edit device information" button.	The app takes the user to a page containing input fields where they can update the device's information.
The user enters the new device information and presses "Edit"	The app updates the device and redirects the user back to the device control page where they can see the new information.

User story: As a smart home owner, I would like to remove a device from my home.	
User interaction	App response
In the list of devices, the user clicks on a specific device.	The app takes the user to a page containing the device's details and settings.
The user taps on the "Edit device information" button.	The app takes the user to a page containing input fields where there is a button to remove the device.
The user presses "Remove device".	The app shows a dialog warning the user of this action and asking if they want to proceed.
The user presses "Yes".	The app removes the device and takes the user to the device list where they will see that the device is gone.

User story: As a smart home owner, I would like to view my devices by location (such as living room, bathroom, bedroom, etc.)	
User interaction	App response
The user presses on the "Rooms" tab on the bottom navigation bar.	The app takes the user to a page containing a list of rooms in their home.
The user presses on a room.	The app takes the user to a page containing a list of all devices in that room.

Design (Level 1)

To fulfill the user stories and use cases defined above, I have designed several screens for the application (see the next page). The four screens at the top of the diagram are the four main screens of the application, and as such they can be reached via the bottom navigation bar. The “Devices” screen contains tabs that take the user to the “Lights” and “Air conditioner” screens. From there, the user can go to each device’s control page, which presents the further option to go to its edit/delete page (the last row).

The pages make use of several widgets such as the bottom navigation bar, the top app bar, the floating action button, sliders, segmented single-choice buttons, and alert dialog boxes.

Comparison with other apps (Level 1)

While designing my application interface, I have taken inspiration from several apps, but the two most influential apps are Google Home and Amazon Alexa, popular smart home apps developed by two major companies in the industry. These apps have inspired a number of UI features in my app, which are:

- ***Having a bottom navigation bar:*** This follows the navigation UI design pattern and allows the app developers to split the content of their application into logical sections. This makes it easier for the user to understand what the main parts of the app are.
- ***Having the option to organize devices by room:*** This makes it easier for the user to manage their homes because they can see all devices in the same room together and control them in one place. This feature is accessible through the bottom navigation bar (see the diagram on the next page).
- ***Having a “favorite” section on the home page:*** This lets users quickly access devices of interest, making the app more convenient to use because the user doesn’t have to scroll through a long list of devices to find what they are looking for. They can immediately start the app and find frequently used devices.

Google Home and Amazon Alexa also have several features that I would like to include in my app, such as creating automated actions and searching and filtering for devices. However, as my app is smaller in scale and this project only allows me enough time to build a working prototype, I will not be implementing these features.

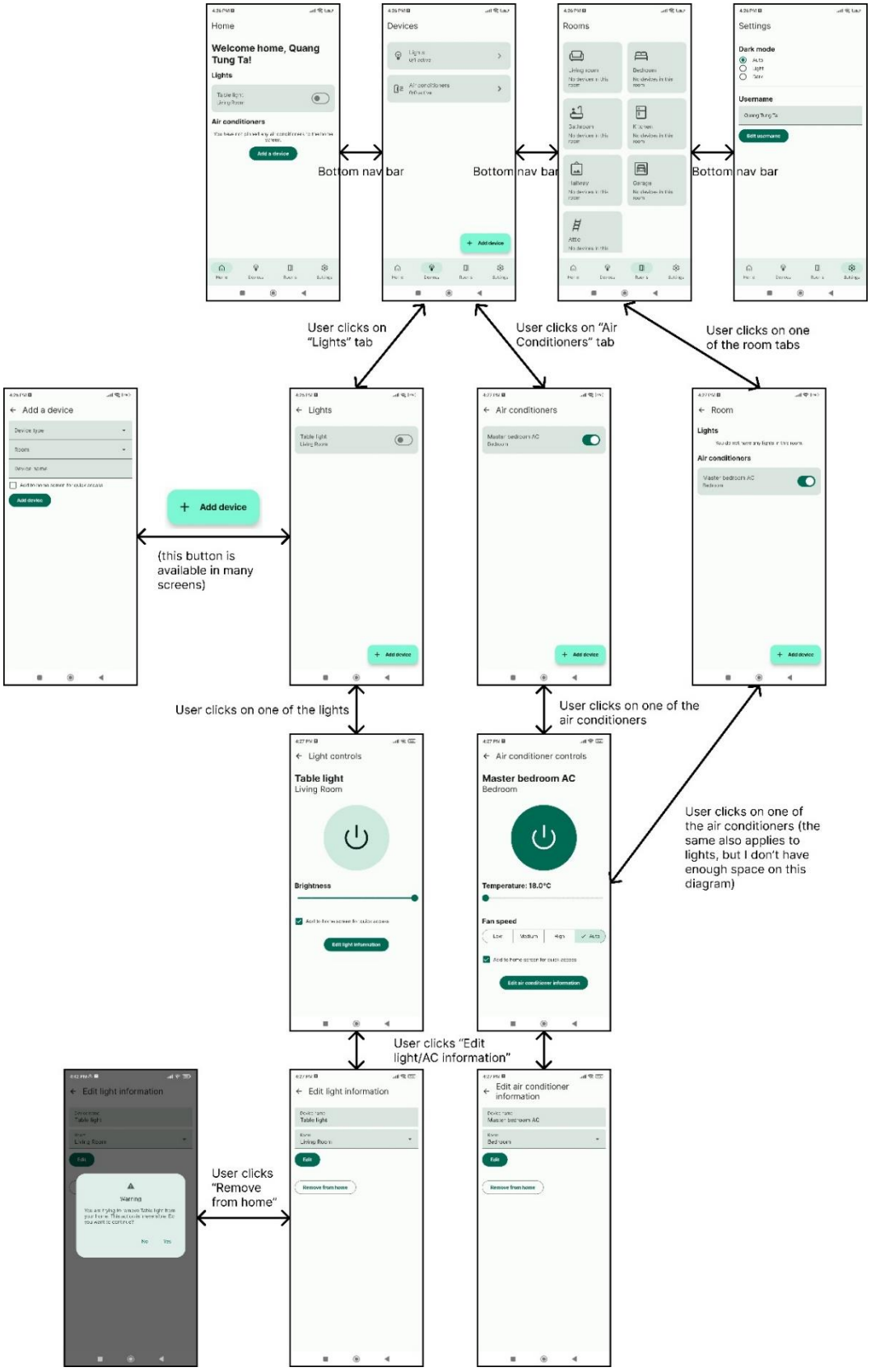

Figure 1: The screens of the custom app.

APIs and architecture (Level 2)

My app is essentially a full-stack mobile application with a front-end and back-end similar to a full-stack web application, with the only difference being that the backend is on the user's device. This is necessary to support all CRUD operations as required by level 2. Following Android's architectural guidelines, I have divided my application into two layers: UI and data.

For the UI layer, I used Jetpack Compose, which is more intuitive than the Views approach and is now Android's recommended way to build user interfaces. It also natively supports Material UI 3, which I have used to consistently style my app.

For the data layer, there are two types of data I want to store: the data of smart devices (lights, air conditioners, etc.) and the application's settings. As the data of smart devices is structured, I choose to use the Room API, which creates and uses a local SQL database on the user's device. Since Room only persists data on the local device, the same data cannot be accessed by the same user on another device. To achieve this, I would have to use a remote database like Firebase Firestore. However, within the scope of this project, using a local database is enough. The database consists of two tables with the following schemas:

Light	
PK	id (Int)
	name (String)
	location (String)
	isOn (Boolean)
	brightness (Float)
	isFavorite (Boolean)

AirConditioner	
PK	id (Int)
	name (String)
	location (String)
	isOn (Boolean)
	temperature (Float)
	fanSpeed (String)
	isFavorite (Boolean)

Figure 2: The Room database schema.

For the storage of application settings, I initially planned on using SharedPreferences. However, after reading Android's developer guides, I decided to use Preferences DataStore instead. It is an implementation of DataStore which reads and stores data in key-value pairs. This storage format is suitable for simple global application settings like dark mode preferences and usernames.

To make the code as organized and scalable as possible, I tried my best to follow Android's architectural practices. These suggest dividing the app into one UI and one data layer as I have described above. The UI layer is responsible for showing the application data on the screen and handling user interactions. This layer is comprised of UI elements (which I have written in Compose) and state holders that hold and expose data, and handle logic (in my app I used ViewModels). The data layer encapsulates business logic and consists of repositories that retrieve data from various data sources. The ViewModels from the UI layer

then interacts with the necessary repositories to fetch data for and update data from the UI. This architecture looks like this:

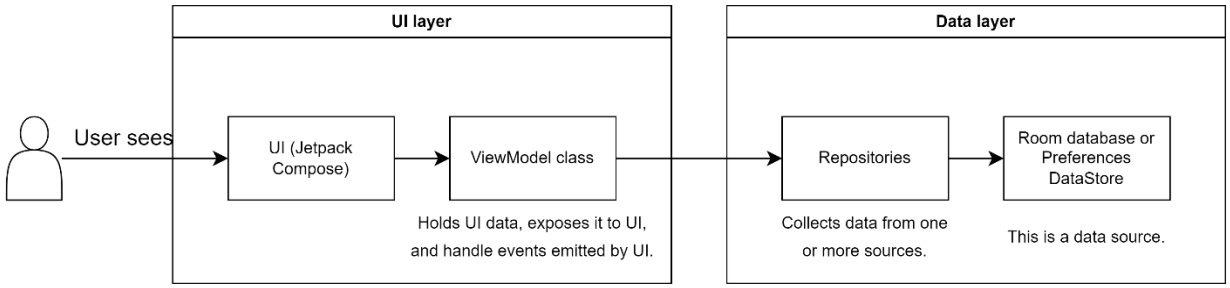

Figure 3: The recommended application architecture.

This model shows that the ViewModel classes are dependent on one or more repositories. To supply them with the dependencies they need, I used manual dependency injection to supply the ViewModel objects with references to repositories when they are created. Each repository is only created once and stored in the Application class. A ViewModel Factory object accesses these repositories and injects them into the ViewModel objects as needed.

The overall architecture of the app looks like this:

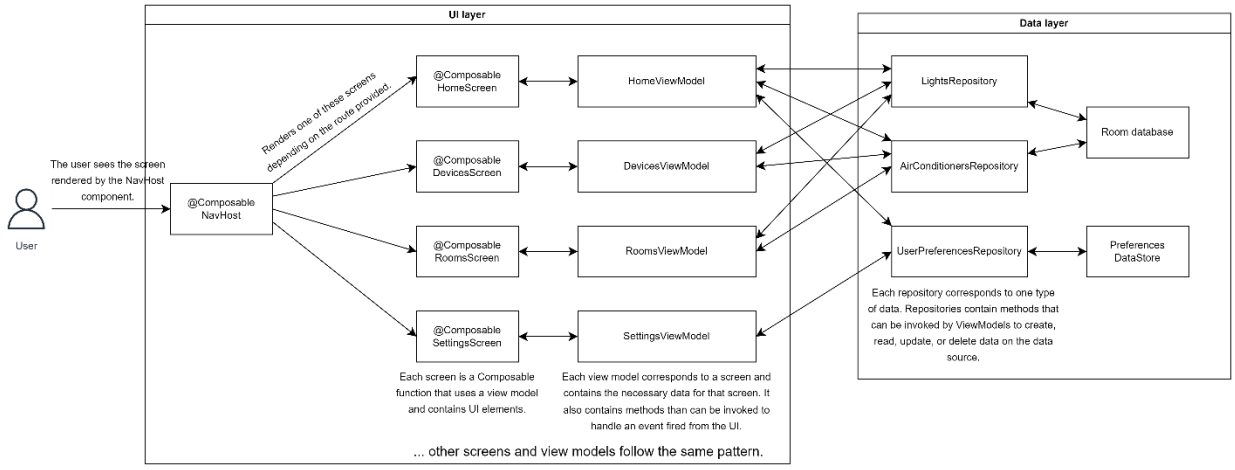

Figure 4: The overall application architecture.

Since the app contains many different screens, a navigation component had to be used to switch between them. This is the NavHost composable, which maps a route to a screen and displays it when the current route matches. Each screen is a Composable function that contains UI elements and uses a view model. This architecture is highly extensible. In the UI layer, all that is needed to add a new screen to the app is to write a Composable function specifying the UI for that screen, a corresponding ViewModel class, and a route in the NavHost. In the data layer, new data sources and repositories can be added without interfering with the logic of other repositories.

Extension on accessibility (Level 3)

As more and more people around the world are using mobile devices, mobile apps must be accessible to be easily used by everyone, including people with disabilities. This is especially important for utility apps like SmartHome, whose purpose is to help people perform everyday tasks more efficiently. The Android framework offers a few guidelines to make apps more accessible, such as making text easier to read, increasing the size of control elements, and providing every UI element with a description (*Make Apps More Accessible | App Quality*, 2024). In an attempt to assess the accessibility of the top Google Play apps, a study by Ballantyne et al. (2018) established a more comprehensive list of guidelines which were grouped into 11 categories: text, audio, video, UI elements, user control, flexibility and efficiency, recognition rather than recall, gestures, system visibility, error prevention, and tangible interaction. As not all of these categories apply to the custom app, it will only focus on the following areas:

- Text: The textual content of SmartHome must be rendered in an adequate size and color.
- UI elements: The UI controls of SmartHome must be labeled, well-colored, well-positioned, and large enough for users to click on.
- System visibility: The operations of the app can be perceived by users.
- Recognition rather than recall: All relevant information to perform a task must be provided on-screen.
- User control: The app must give users enough time to process content on the screen, and should follow the users' settings such as device orientation and dark mode preferences.

Thankfully, most of these criteria are already satisfied simply by using Jetpack Compose. This is because Jetpack Compose is bundled with Material design by default, and its native components already follow these design guidelines. For instance, Material automatically maps suitable colors to the background and text of a component to ensure adequate contrast. Compose and Material also supports orientation changes and dark mode preferences by default, so minimal code has to be written. The system visibility criteria, which is not supported by default, is achieved by using snack bars that inform the user of the state of the app every time they perform an operation such as turning on a device.

In addition to satisfying the above design guidelines, SmartHome also improves its accessibility by integrating seamlessly with TalkBack, Android's screen reader. TalkBack works with every Android app by default, but SmartHome UI's semantics have been customized so that it is clearer to the user what they are interacting with. For example, without any modification, focusing on the switch in the figure below makes TalkBack announce "On, switch, double tap to toggle". However, after customization, it reads "On, switch, double tap to turn off Two-way AC", which is much clearer to the user what the control does.

Figure 5: The highlight switch will be announced differently by TalkBack thanks to customization.

Limitations and areas of improvement

Due to time constraints, this project has several areas that could be improved. First, it does not support the full suite of smart home devices, which can include doors, cameras, TVs, etc. It is limited to only lights and air conditioners, which is unlikely to be enough for real users. Second, the app does not have an authentication feature, meaning that anyone can access and control the devices. In addition, this prevents users from being able to manage different homes on the same device. Third, the app currently uses a local database, which prevents users from managing the same home on different devices. It also causes the home data to be deleted when the app is deleted. A real application would need a remote database like Cloud Firestore to share data across different devices.

References

Ballantyne, M., Jha, A., Jacobsen, A., Hawker, J. S., & El-Glaly, Y. N. (2018). Study of Accessibility Guidelines of Mobile Applications. *Proceedings of the 17th International Conference on Mobile and Ubiquitous Multimedia*, 305–315. https://doi.org/10.1145/3282894.3282921

Make apps more accessible | App quality. (2024, January 3). Android Developers.

https://developer.android.com/guide/topics/ui/accessibility/apps

Appendix

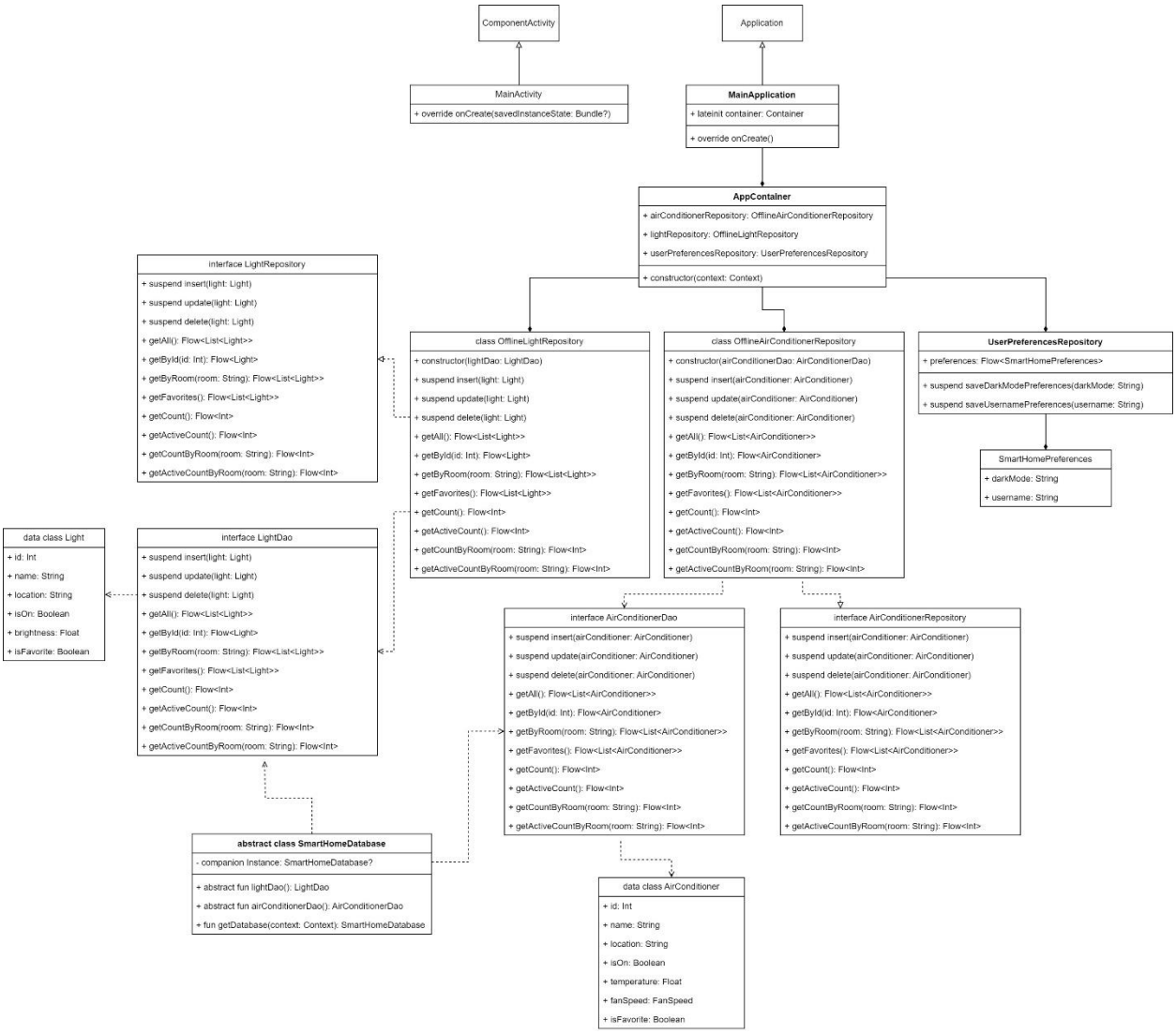

Figure 6: Class diagram of SmartHome.