Group 1

Peridot: Connecting friends, families & teams

Written by:
LEUNG, Tsz Ching (20588530)
TSANG, Chin Yung Virginia (20762977)
WANG ZEKAI (20764298)

Date: December 20, 2023

Course Instructor: Professor SZETO, Lok Chun

Word Count: 1,123

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong, China

Contents

1 Background **2**

2 Objectives **2**
2.1 Waterfall Development 2

3 Design Specification **4**
3.1 Use Case Diagram 4
3.2 User Journey 6
3.3 Class Diagram 7
3.4 Architecture 8
3.4.1 Login System 8
3.5 User Interface 10
3.6 Database Schema 11

4 Implementation **12**
4.1 Database and Room 12
4.2 PDF export 13
4.3 Email Intent 13
4.4 Scheduler 14
4.5 Group Setting 14

5 Testing **15**
5.1 Test Cases 15
5.2 Test Report 15

6 Planning **16**
6.1 Gantt Chart 16
6.2 Division of Labor 17

7 Possible Improvments **17**

8 Conclusion **18**

1 Background

In today's digital age, communication tools are abundant, with each promising faster ways to connect. However, a significant gap remains in the quality of connections we establish. The rise of social media and instant messaging has led to an era of rapid, often superficial interactions, leading to what some have termed "loneliest generation Z". Traditional methods like calls and texts, while valuable, can sometimes fall into repetitive patterns, missing out on more introspective conversations. This situation underscores the need for a solution that bridges this divide, offering both the convenience of digital communication and the depth of personal interaction. Our proposed app, Peridot, is a platform for individuals to engage in more profound conversations among friends, families, and teams through group newsletters - collaboratively written articles issued for a group of friends.

2 Objectives

Peridot brings depth to conversations by private group newsletters for friends, families and teams. The primary objective of Peridot is to provide a space for deeper, more meaningful connections among friends, families, and teams. Unlike conventional communication tools that emphasize speed and volume, our app will prioritize quality and depth. From the project's outset, we prioritized security and user experience as fundamental aspects. Security measures are in place to protect users' personal data and group conversations. A user-friendly interface is crucial to foster a strong sense of community among users.

2.1 Waterfall Development

For our project, we have chosen the Waterfall Development Method in Fig. 1 for two primary reasons. Our primary goal with Peridot is to showcase our innovative ideas for a social application. We're not aiming for a fully-fledged product but rather a demonstration of our software development capabilities. This includes both our coding expertise and our ability to select and implement the right project management framework. The second reason is that this is a one-time submission. Other methodologies like the Agile Development Methodology require hastily and repeatedly rectifying the application. Applying the Waterfall Method will let us concentrate on unfinished tasks without constantly revisiting completed ones.

1. User Requirement Analysis: We understand the urgent need for people to build

stronger communication with their social group. We have identified college students as a key demographic. Furthermore, it is also important to take their parents into consideration. They represent the people who might be invited to answer the questions, and some parents are not experts in using mobile applications. By addressing the needs of both groups, we aim to create an app suitable for users of all ages.

2. System Architecture Design: Given that this is an Android mobile software project, we'll develop a front-end UI for user interaction and a database for data storage and logical computations.

3. Software Programming: The front-end development kit is fixed to Android Studio using Kotlin. For the back-end server and database, we will stick to the instructions of the course.

4. Software Testing: Unit testing will be conducted on individual code segments during the programming phase. Subsequently, system testing will ensure the entire system functions cohesively, especially the interactions between the front-end and the database

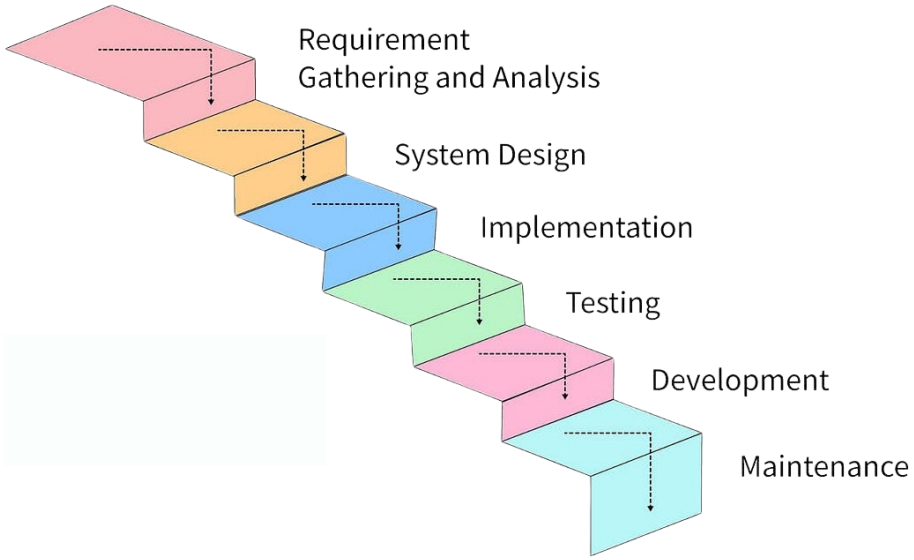

Figure 1: The Waterfall model

3 Design Specification

3.1 Use Case Diagram

The functionalities of the Peridot app are examined through the perspective of what actions users can perform within the system. These functionalities are categorized into four primary use cases: Create and Manage Groups, Manage and Compose Own Letter, View Group Letter Issues, and Manage User Account, as shown in Fig. 2

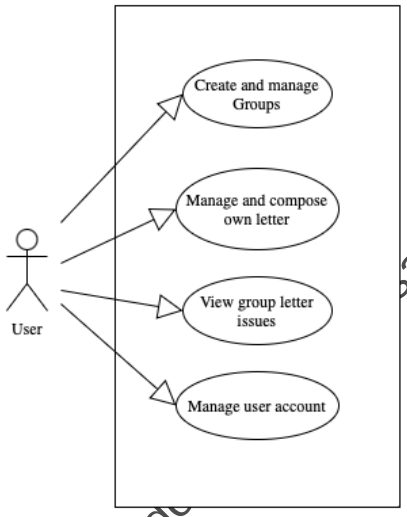

Figure 2: Use Case diagram for Peridot

The `Create and Manage Groups` use case involves users establishing new groups and overseeing existing ones. This encompasses the initial setup of a group, where users can define group details, and the ongoing management, which includes inviting new members via a shared invitation code by email. Users can also adjust settings like the frequency of letter publications and access a question bank for discussion prompts.

In `Manage and Compose Own Letter`, users compose and manage their responses to group letters. This use case allow users to draft, save, and edit their contributions to the group's letters. Users can also save progress and clear all responses.
The `View Group Letter Issues` use case addresses the retrieval and review of pub-

lished group letters. This functionality allows users to access past issues, serving as a digital archive of the group's exchanges. The ability to export these letters in PDF provides users with a means to preserve their collective memories outside the app.

Lastly, the `Manage User Account` use case encompasses the administrative aspects of a user's interaction with the app. This includes basic account setup, login, and personal settings management.

3.2 User Journey

Refer to Fig. 3. Users begin by logging in or registering an account in Peridot. Once logged in, users can create or join groups, or view their current groups. To join an existing group, a user needs to enter a unique verification code, which is obtained from an existing member of the group. Alternatively, users can create a new group by the [+] button on the bottom right of the user screen. The process of creating a group includes setting up group details and inviting friends to join.

To manage groups in the group's settings, users can access and share the group's invitation code using email, enabling the addition of new members. The frequency of letter publication is adjustable, and users have access to a question bank for prompting discussions. Group settings also allow users to view group members and their respective roles.

Responding to questions for upcoming group letters is the main activity within the app. Users select the 'reply now' option to access the questions for the next issue. The app provides features to save progress on responses, ensuring that users can complete their input in multiple sessions. A 'clear all' option is also available, offering users the flexibility to restart their responses if desired. The app reloads previously saved responses when users return to complete their entries. Customization of questions happens separately from the current issue's responses. It allows users to tailor the conversation topics within their group.
Once the letter is published, users can view it under the Past Issue section. This section acts as an archive, storing previous letters written by group members. Users have the option to export these letters in PDF format, providing a means to preserve past conversations.

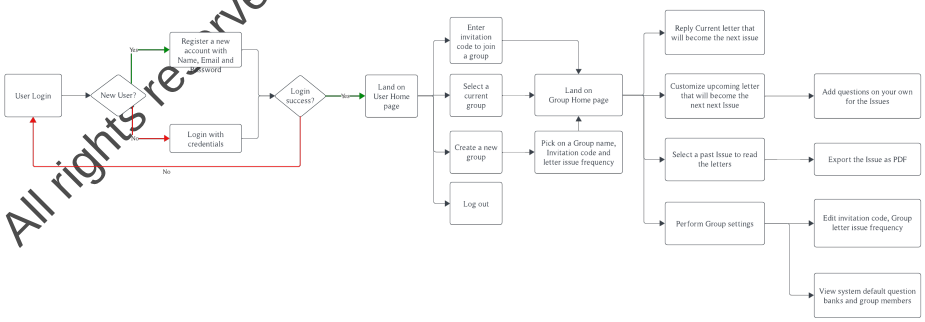

Figure 3: User Journey of Peridot

3.3 Class Diagram

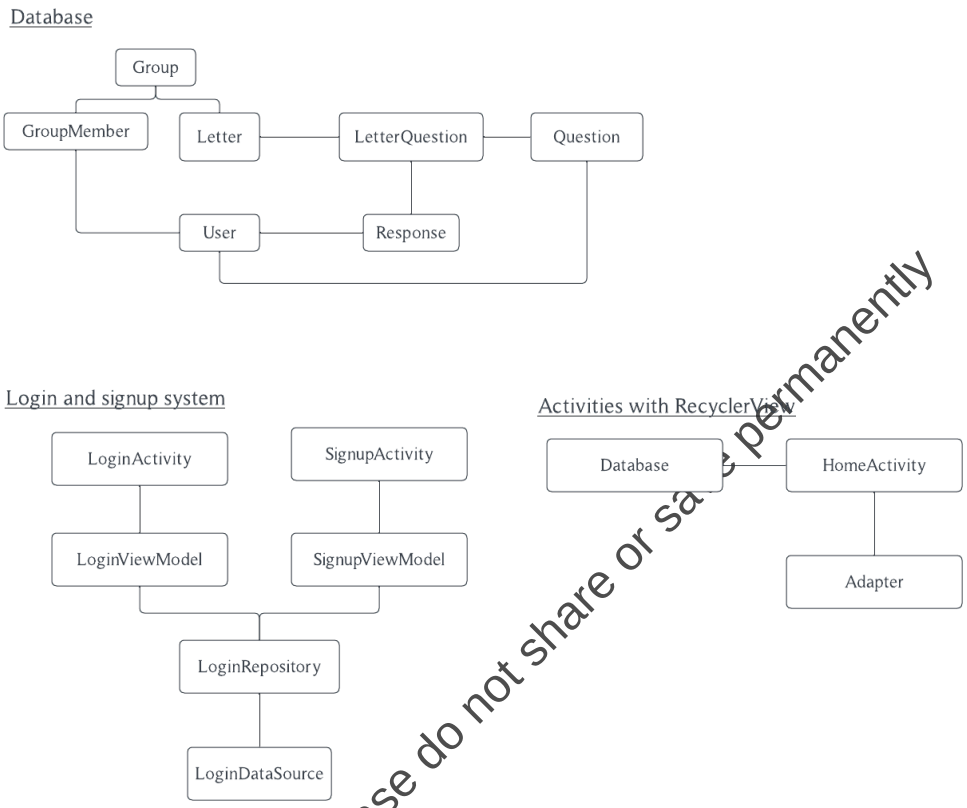

Figure 4: Class diagram for Peridot

3.4 Architecture

Peridot built on Android Studio follows the recommended app architecture that includes the use of components such as ViewModels, LiveData, and the Room database. ViewModel is used to manage UI-related data in a lifecycle-conscious way, allowing data to survive configuration changes such as screen rotations. LiveData is an observable data holder class that works well with ViewModel, ensuring that the UI matches the app's data state and allows for reactive UI updates. The Room database is an abstraction layer over SQLite, simplifying database access while harnessing the full power of SQLite. These components are integral to the app's Model-View-ViewModel (MVVM) architecture.
Each activities in the app serves a specific function in the user interface. For instance, activities like `LoginActivity`, `SignupActivity`, `MainActivity`, and `SplashActivity` manage different aspects of user interaction, from initial login to the main user interface. Specialized activities such as `PDFActivity` and `LetterDisplayActivity` are dedicated to displaying PDFs and letters. Fig. 5 shows an overview of the app's architecture.

Navigation within the app is facilitated through an intent filter in the `MainActivity`, serving as the primary entry point. This is complemented by a series of activity transitions, allowing users to navigate through different functionalities such as group creation and settings management. The app's focus on group and letter management is evident in activities like `GroupHomeActivity` and `CustomizeLetterActivity` and `WriteQuestionAnswerHomeActivity`, which likely offer features for managing groups and customizing letter content.

3.4.1 Login System

The architecture of the login and signup system in Fig. 6 in the provided Android application is designed in a modular manner. This system, built using the Model-View-ViewModel (MVVM) pattern, separates the user interface (UI) from the business logic.

At the core of the system, there are two primary activities (View): `LoginActivity` and `SignupActivity`. These activities serve as the entry points for users to interact with the application, providing the UI for users to enter their credentials. `LoginActivity` manages the login process, while `SignupActivity` handles new user registrations. Each activity is backed by its corresponding ViewModel - `LoginViewModel` for

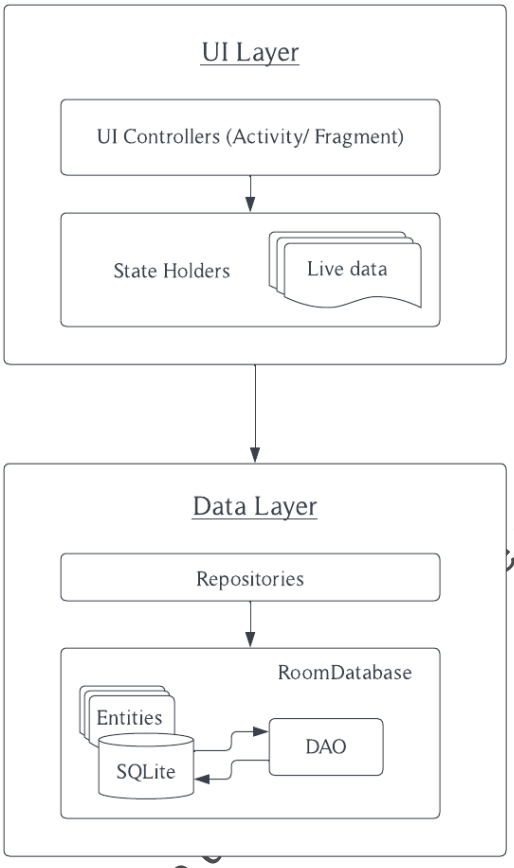

Figure 3: Peridot app archetecture

`LoginActivity` and `SignupViewModel` for `SignupActivity`. These ViewModels are responsible for processing the data entered by users. They contain the logic for validating user input, such as checking the correctness of email formats and password strength.

The ViewModels interact with `LoginRepository`, a central class that acts as a bridge between the data source and the ViewModels. `LoginRepository` is tasked with managing the authentication data flow. It communicates with LoginDataSource, which is likely responsible for the actual authentication operations, such as querying a database or interacting with a remote server. For managing the state and results of the authentication process, the system utilizes data classes like LoginResult, LoggedInUserView, and LoginFormState. LoginResult encapsulates the outcome of the authentication

attempt, whether it's a success, rejection, or error. LoggedInUserView is used to represent the authenticated user's information post-login, and LoginFormState holds the current state of the login/signup forms, including any validation errors.
LiveData objects within the ViewModels (loginFormState and loginResult) are used to observe and react to changes in the authentication process. This reactive approach ensures that the UI is consistently updated with the latest state, providing real-time feedback to the user. The system also employs LoginViewModelFactory and SignupViewModelFactory for instantiating the ViewModel objects. This use of the Factory pattern is particularly beneficial when the ViewModels have dependencies that need to be injected, such as the `LoginRepository`.

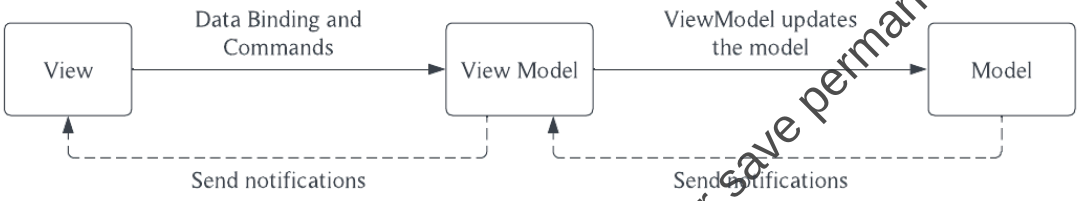

Figure 6: ModelModel-View-ViewModel archetecture

3.5 User Interface

As a mobile application, Peridot is dedicated to providing a simple and clean user interface for user-friendly experiences as in Fig. 7. Peridot's user interface theme takes inspiration from the Letter Social Mobile App template created by ALMAX Design Agency , aiming to create an interface that is characterized by its simplicity and cleanliness.

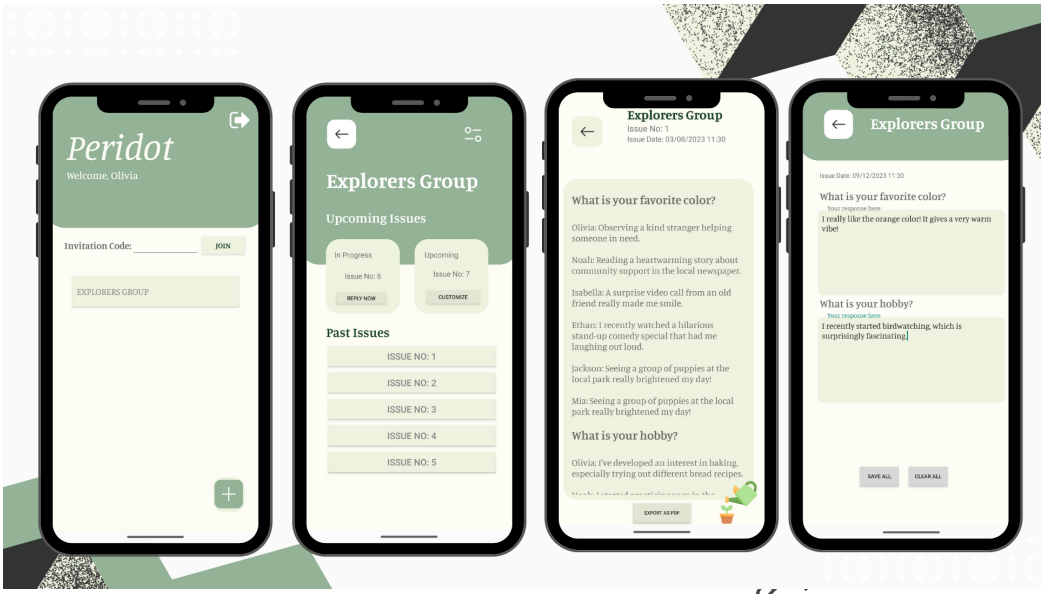

Figure 7: User Interface for Peridot

3.6 Database Schema

The schema in Fig. 8 interconnects several key tables: `users`, `groups`, `group_members`, `letters`, `questions`, `letter_questions`, and `responses`. The `users` table is fundamental for managing individual user accounts. This table is linked to the `groups` table, which manages different user groups, each with unique attributes like group name and newsletter frequency.

Roles within these groups are defined in the `group_members` table, establishing a relationship between individual users and their respective groups, along with their roles. This setup is for role-based access and control within the app, a future feature to be implemented.

Content creation is primarily handled through the `letters` and `questions` tables. The `letters` table stores information about the group newsletters, while the `questions` table holds the questions that are included in these newsletters. The `letter_questions` table serves as a junction between the two, linking specific questions to their respective newsletters. The `responses` table records user responses to the questions in the newsletters, creating a link between user input and the content of the newsletters.

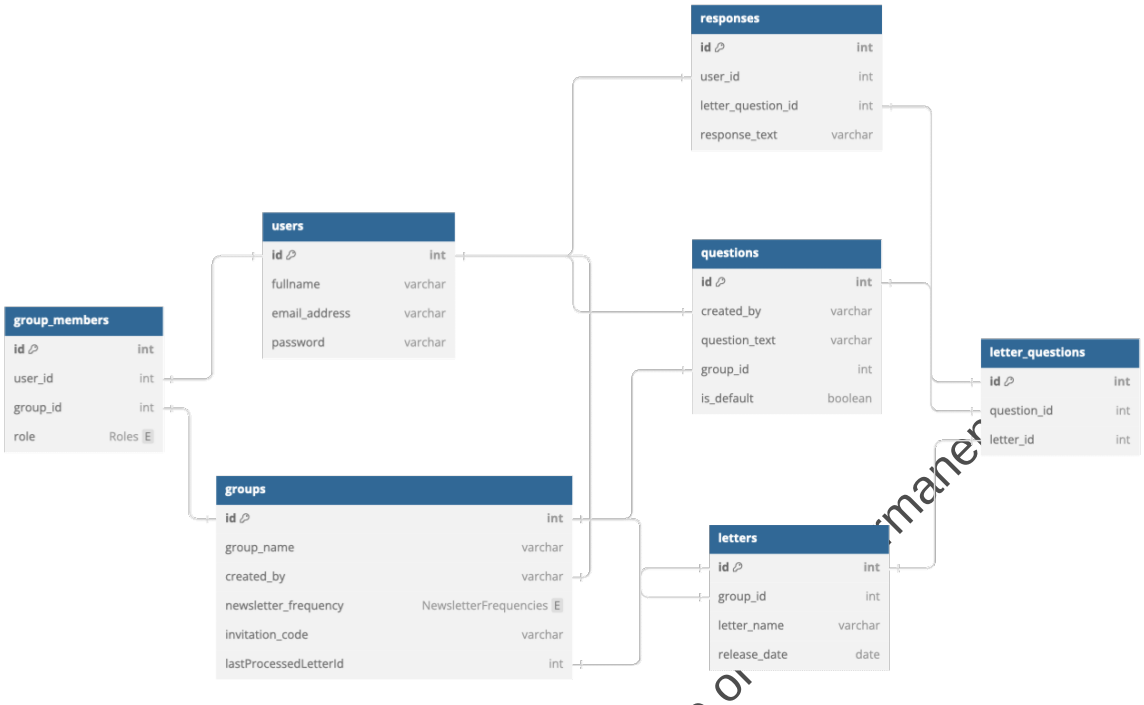

Figure 8: Database schema

4 Implementation

4.1 Database and Room

The database is constructed using Room, a persistence library that provides an abstraction layer over SQLite. Upon initialization, the `appDatabase` class includes a companion object that manages the instantiation of the database. It uses a singleton pattern to ensure that only one instance of the database is created. The `buildDatabase` method initializes the database, and checks are performed to ensure that the database is only populated with initial data when it's not already initialized.

The database encompasses several entities like `User`, `Group`, `GroupMember`, `Letter`, `Question`, `LetterQuestion`, and `Response`. The database uses `@Dao` interfaces, such as `UserDAO`, `GroupDAO`, `GroupMemberDAO`, `LetterDAO`, `QuestionDAO`, `LetterQuestionDAO`, and `ResponseDAO`. These DAOs declare the necessary methods to access the database, including insert, update, delete, and complex queries. The use of DAOs promotes a clean API for the app to interact with the database and ensures separation of concerns.

`UserDAO` focuses on user-related operations. It includes methods for inserting new

users, querying user counts by name, retrieving user details by ID, and checking login credentials. This DAO is central to managing user authentication and profile information.`GroupDAO` handles all operations related to groups. It features methods for inserting and managing groups, retrieving group details by various parameters, and accessing group-related data such as upcoming and past letters, and group frequencies.`GroupMemberDAO` manages group member data. It includes functionalities to add, update, and remove members from groups, as well as queries to check member existence and retrieve member details.`LetterDAO` manages letters. It includes methods for inserting letters, updating them, and fetching letter details based on various criteria.`QuestionDAO` deals with the operations related to questions in the app. It includes methods for managing (adding, updating, deleting) questions and fetching questions based on different criteria.`LetterQuestionDAO` keeps the relationship between letters and questions. It includes methods to manage these relationships and queries to fetch specific letter-question pairings.`ResponseDAO` handles user responses. It includes methods for inserting, updating, and deleting responses, as well as fetching responses based on different criteria.

The `Converters` class is used to handle the conversion of custom data types (like `LocalDateTime` and `Roles`) to and from database-compatible types. This is essential for storing complex data types in the SQLite database.

4.2 PDF export

The `PdfGenerator` class is responsible for converting the newsletter content, presented in a user interface layout, into a PDF document. This process involves measuring and rendering the layout onto a `Bitmap`, which is then transposed onto a `Canvas` linked to a `PdfDocument`. The resulting PDF is a direct representation of the user interface. The class manages the storage of the generated PDF in an app-specific external directory.

4.3 Email Intent

The implementation of the email function within the `ScheduleFragment` is structured as an Android `Fragment`. In the `onCreateView`, various UI elements are initialized, including a button `share_btn`. When the button is clicked, the app constructs an email intent using `Intent.ACTION_SENDTO`, specifying that the intent's action is to send an email. The email content is formatted in HTML. The intent is configured with the `Uri.parse("mailto:")` method, indicating that the data being passed is an email address. The `Intent.EXTRA_SUBJECT` and `Intent.EXTRA_TEXT` fields are used to set the subject and body of the email. Upon triggering this intent, the app

attempts to start an activity that can handle the email sending process. This action opens the user's default email client with the pre-filled subject and body, ready to be sent to potential group members.

4.4 Scheduler

The `LetterCheckWorker` class, derived from Android's `Worker` class manages newsletter issuing. It operates periodically, interfacing with the app's database through `GroupDAO` and `LetterDAO`. This class checks for updates or necessary actions in the newsletter lifecycle, utilizing a `LetterManager` instance and a `PdfGenerator` to manage and generate newsletters, respectively. `LetterManager`, as the starter of newsletter updates, interfaces with the database to monitor and manage the lifecycle of each newsletter. It is tasked with identifying the current and upcoming newsletters for each group and triggers the creation of new newsletter issues based on predefined scheduling time in the group settings.

4.5 Group Setting

The `GroupSettingActivity` extends `AppCompatActivity`, indicating the use of Android compatibility features. Upon creation, the activity sets its content from a layout resource (`activity_group_setting`). The activity retrieves a group ID passed through an `Intent`, signifying the specific group for which settings are being adjusted.

The activity manages three primary buttons (`scheduleBtn`, `QBBtn`, `memberBtn`) which are responsible for navigating between different fragments. Fragment transactions are used extensively to switch between `ScheduleFragment`, `QuestionBankFragment`, and `MembersFragment`. This approach allows for a seamless and responsive user experience, as different aspects of the group setting are modularly separated into fragments. `MembersFragment` handles displaying the members of a group. It uses a `RecyclerView` to list members, showcasing a modern approach to displaying dynamic lists in Android. The fragment interacts with the database to retrieve and display member details, indicating integration with backend data structures. `QuestionBankFragment` is focused on managing the question bank for a group. It similarly utilizes a `RecyclerView` for displaying questions. The fragment also implements popup windows for editing or adding questions. The `ScheduleFragment` is another crucial fragment within the `groupsetting` package, designed to manage scheduling aspects for a group. Like the other fragments (`MembersFragment` and `QuestionBankFragment`), it extends the `Fragment` class, adhering to Android's recommended practices for modular UI components. Upon creation, the `ScheduleFragment onCreateView` method inflates the fragment's layout from `fragment_schedule`, establishing the user interface for the

scheduling functionalities. This fragment is responsible for handling scheduling-related tasks. These could include setting up the frequency of group letters, scheduling meetings or discussions within the group, or any other time-related functionalities that the app may offer to its group users.

Testing

Testing is crucial for our system's to ensures the correctness of our application's behavior and helps identify and address any possible issues and errors. In order to provide a consistent and error free user experience, we have conducted testings on our Activities utilizing `AndroidJUnit4` as the testing framework for our Android Application.

5 Testing

5.1 Test Cases

Our testing strategy is designed to validate the functionalities of individual activity and their seamless transitions within the app. We have placed particular emphasis on the navigation flow between activities and tested each pathway to guarantee that the app's navigation aligns with our intended user experience. For individual activities, our test cases are tailored to confirm that ata is correctly displayed and key functionalities, such as item selections and button clicks are performing as anticipated.

Utilizing `Espresso`, an Android testing framework built on top of `AndroidJUnit4`, we have crafted and executed automated UI tests that simulates user interactions such as button clicks and text input. In our test case, we utilizes Espresso's `ViewMatchers` and `ViewActions`, to first find an element within the current view hierarchy, similar to `findViewById` in activities. Once `Espresso` matches a view, it can perform interactions such as scrolling and typing to simulate user actions. Lastly, `viewAssertions` are used to verify the state of the view and check various aspects, such as if a view is displayed or if a text view contains a certain text.

5.2 Test Report

In order to ensure thorough code testing, we have incorporated `Jacoco`, a widely-used code coverage library, to generate testing summaries. By leveraging `Jacoco`, we gain

valuable insights into the effectiveness of our test suite. Currently, our test passing rate stands at a 40%. We understand it is not a promising number, but all of the test cases passes if we run them individually. One possible reason we think of is that, during the whole-testing of Jacoco, the sequence of activities are messed up, and the testing units lost the control of the current activity, resulting the "assertion statement" matches the wrong activities, giving a "false" of this test.
However, since UI test is only part of the testing. During coding, we have tested and fixed many logical related bugs, so we believe that we have successfully achieve a satisfactory testing performance.

6 Planning

6.1 Gantt Chart

The provided Gantt chart (Table 1) illustrates the comprehensive project planning, encompassing documentation, design and setup, implementation, and testing phases.

Tasks	Oct			Nov					Dec
	1	2	3	4	5	6	7	8	9
Proposal									
UI Design									
Database Setup									
Frontend Implementation									
User Functionalities Implementation									
Group Functionalities Implementation									
Edit Letter Functionalities Implementation									
View Letter Functionalities Implementation									
Testing									
Final Report									
Presentation Preparation									

Table 1: Gantt Chart

6.2 Division of Labor

Our team has defined distinct roles and responsibilities for each team member. The detailed breakdown of responsibilities can be found in Table 2, which outlines the allocation of duties for each team member.

Tasks	Jason	Sarena	Virginia
Proposal	A	R	R
UI Design	R	R	A
Database Setup	R	A	R
Frontend Implementation	R	R	A
User Functionalities Implementation	A	R	R
Group Functionalities Implementation	R	A	R
Edit Letter Functionalities Implementation	R	A	R
View Letter Functionalities Implementation	R	R	A
Testing	A	R	R
Final Report	R	A	R
Presentation Peparation	R	R	A

Table 2: Division of Labor

7 Possible Improvments

The current state of the Peridot app allows all users equal rights across various functionalities, including group creation and management, letter composition and management, viewing group letters, and managing user accounts. While this approach simplifies the user interface and experience, it presents limitations in terms of role-based access and administrative control. It is possible that we introduce an administrative role that distinguishes user rights.The existing database design in Peridot allows implementation of administrator roles.

- Currently, any group member can adjust the frequency of letter publication. With the proposed improvement, only administrators would have the authority to make these changes.

- At present, none of the members can modify the set of app provided questions used to prompt discussions. Granting only administrators the right to change these questions will ensure consistency and relevance of conversation topics.

8 Conclusion

In the journey of developing Peridot, we have navigated through the multifaceted landscape of modern digital communication, addressing the pressing need for deeper and more meaningful interactions in the digital age. Peridot stands as a testament to our commitment to bridging the gap between the convenience of digital communication and the depth of personal interaction.

From the outset, our focus has been on creating an app that not only prioritizes security and user experience but also fosters a sense of community. The Waterfall Development Method, chosen for its streamlined approach, allowed us to concentrate our efforts on showcasing our innovative ideas and software development capabilities.

Our implementation strategy encompassed a robust architecture, integrating various functionalities from group management to content creation. The use of the Room Database and adherence to the MVVM architecture ensured efficient data handling and responsive user interactions. The PDF generation and email functionalities added layers of practicality.

In retrospect, the development of Peridot has been a fulfilling journey. We have successfully created a platform that not only meets the contemporary demands of digital communication but also enriches it with depth and significance. As we look to the future, the potential for further enhancements, such as the introduction of administrative roles, holds the promise of making Peridot an even more versatile platform.