Orphanage External Activity System

Course Code: CSE336

Course Name: Software Design Patterns

Senior 2 - Computer Engineering and Software Systems

Hope Home

Team members:

Adham Hatem	**20P8384**
Ahmed Ibrahim Eldera	**20P2701**
Mohammed Ayman Gharib	**20P1776**
Sherwet Mohamed	**20P8105**
Yassin Khaled Abd ElSamie	**20P2668**
Yousef Fayez Ibrahim	**2101616**

Submitted to :

Dr: Ayman Ezzat

Eng: Ahmed salama

System Description:

This system is designed for orphanages seeking to host external events and collaborate with external entities, such as volunteers and donors, who are not part of their internal management system. The platform facilitates seamless interaction between the orphanage and external contributors, enabling efficient event management, volunteer coordination, and donation tracking.

Future integration with the internal orphanage management system is possible, allowing for smooth data and information sharing if required.

User Stories

Admin

- **As an admin**, I want to create, manage, and track events so that I can efficiently organize them.
- **As an admin**, I want to assign volunteers to tasks to help execute the event smoothly.
- **As an admin**, I want to track donor information, including their contact details and donation history, so that I can maintain accurate records.
- **As an admin**, I want to store and manage beneficiary information, including their needs, so that I can track the impact of the programs provided.
- **As an admin**, I want to evaluate program impacts on beneficiaries so that I can improve future initiatives.
- **As an admin**, I want to track volunteer hours and generate certificates so that I can acknowledge their contributions.
- **As an admin**, I want to send email notifications and SMS updates to donors, volunteers, and beneficiaries so that they stay informed.
- **As an admin**, I want to generate detailed reports on donations, events, and volunteer contributions so that I can analyze performance and compliance.

Donor

- **As a donor**, I want to donate using various methods (PayPal , Visa …etc) so that I can contribute in a way that is convenient for me.
- **As a donor**, I want to receive receipts and tax-related documents for my donations so that I can use them for tax purposes.
- **As a donor**, I want to view a history of my donations so that I can track my contributions.

Volunteer

- **As a volunteer**, I want to view upcoming events and tasks so that I can plan my schedule.
- **As a volunteer**, I want to track the total hours I've worked so that I can monitor my contributions.
- **As a volunteer**, I want to view a certificate for my volunteer hours so that I can showcase my efforts.

General User

- **As a user**, I want to be able to securely sign up and login with an easy to understand UI and readable and meaningful errors if I do something wrong.
- **As a user,** I want the option to edit my info in the future if I want.
- **As a user,** I want to receive messages and notifications in my inbox inside the app.

Non-Functional Requirements

1. The system shall ensure data security and compliance with privacy regulations.
2. The system shall provide a user-friendly interface.
3. The system shall support high availability and scalability to handle a growing number of users and donations.
4. The system shall generate reports and logs for auditing purposes.
5. The system shall ensure compatibility across all mobile devices (IOS & Android).
6. The system shall maintain response times for common operations (Assuming good internet connection).

Contribution Table:

Ahmed El-Dera	Template,Facade,Factory,Proxy
Mohamed Ayman	Command,Iterator,Builder
Yousef Fayez	Strategy,State,Iterator
Yassin Khaled	Decorator,Adapter,Class Diagram
Sherwet Mohamed	Observer,Decorator,System design
Adham Hatem	Strategy,Singleton,Class Diagram

GitHub Link:

https://github.com/Ahmed-Eldera/Orphange

Video Link:

https://drive.google.com/drive/folders/10cennYOf9hR4lxudtAldTYRNixxFN1RW?usp=share_link

Class Diagram draw.io Link:

Draw Class Diagram

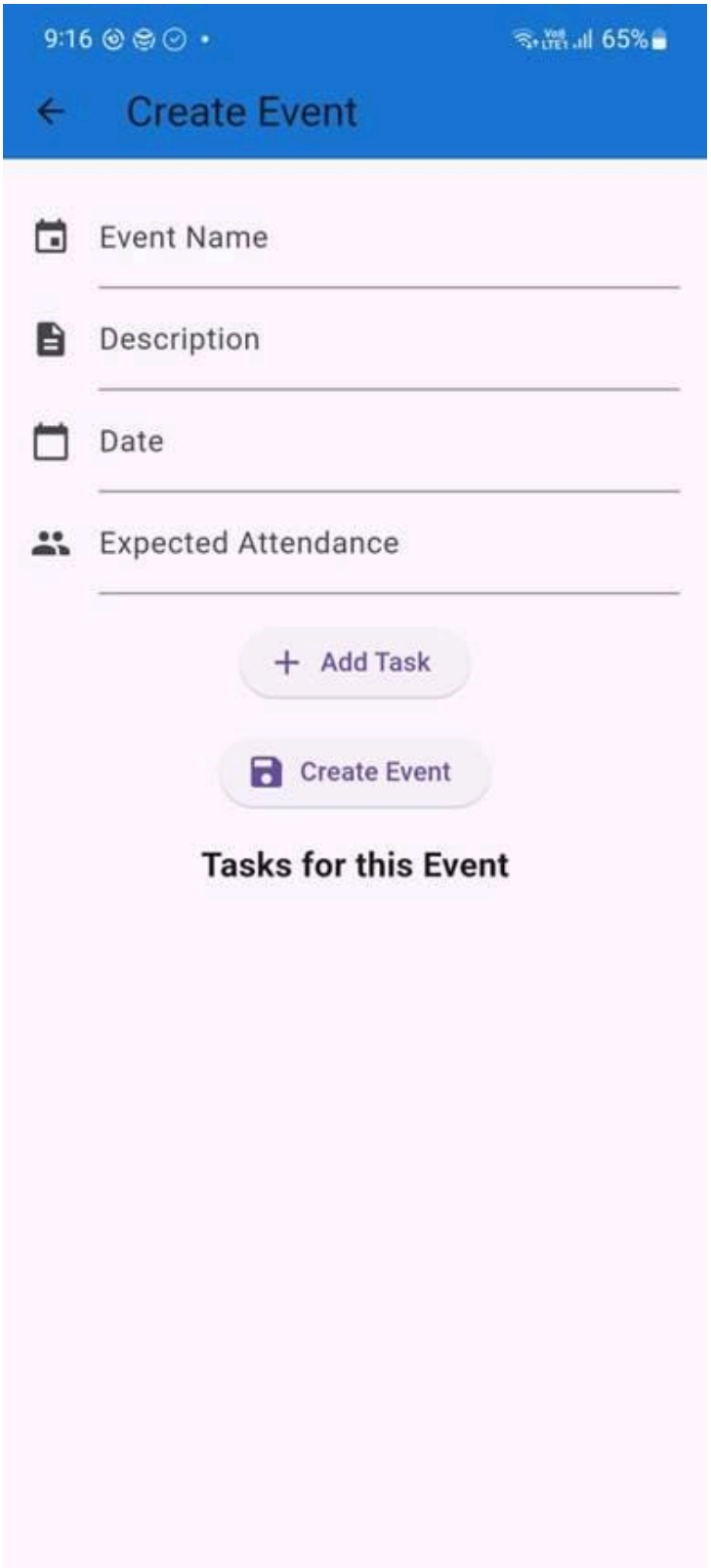
Design patterns in code & screenshots :

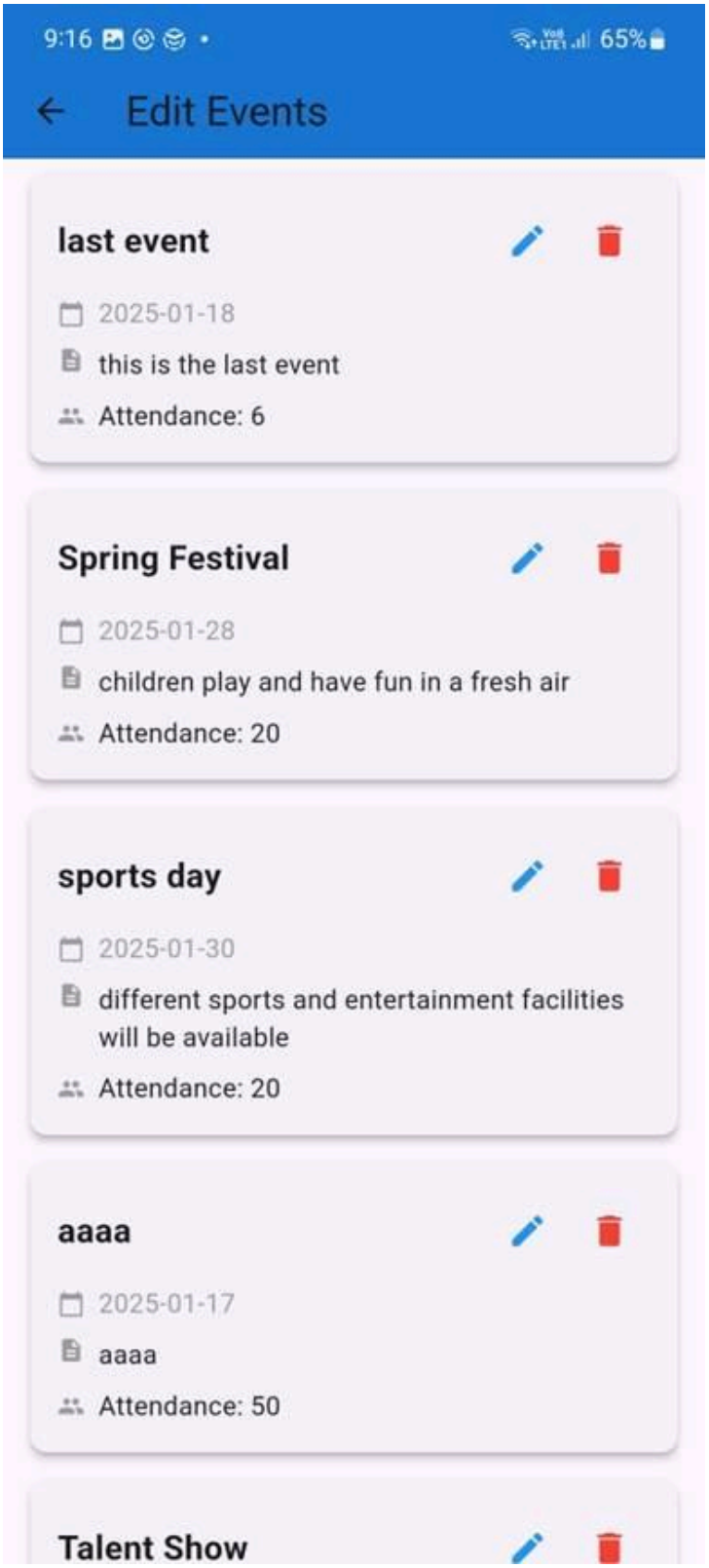
Command design pattern:

```dart
class CreateEventCommand implements Command {
  final EventController eventController;
  final DocumentReference eventDocRef;
  final Event event;
  final List<Task> tasks;

  CreateEventCommand({
    required this.eventController,
    required this.eventDocRef,
    required this.event,
    required this.tasks,
  });

  @override
  Future<void> execute() async {
    // Save the event
    await eventController.saveEvent(eventDocRef, event);

    // Save all tasks associated with the event
    for (var task in tasks) {
      task.eventId = event.id; // Assign event ID to each task
      await eventController.saveTask(task);
    }
  }
}
```

```dart
class DeleteEventCommand implements Command {
  final EventController eventController;
  final String eventId;

  // Constructor to accept eventController and eventId
  DeleteEventCommand({
    required this.eventController,
    required this.eventId,
  });

  @override
  Future<void> execute() async {
    // Delete the event
    await eventController.deleteEvent(eventId);

    // Delete associated tasks
    await eventController.deleteTasksByEventId(eventId);
  }
}
```

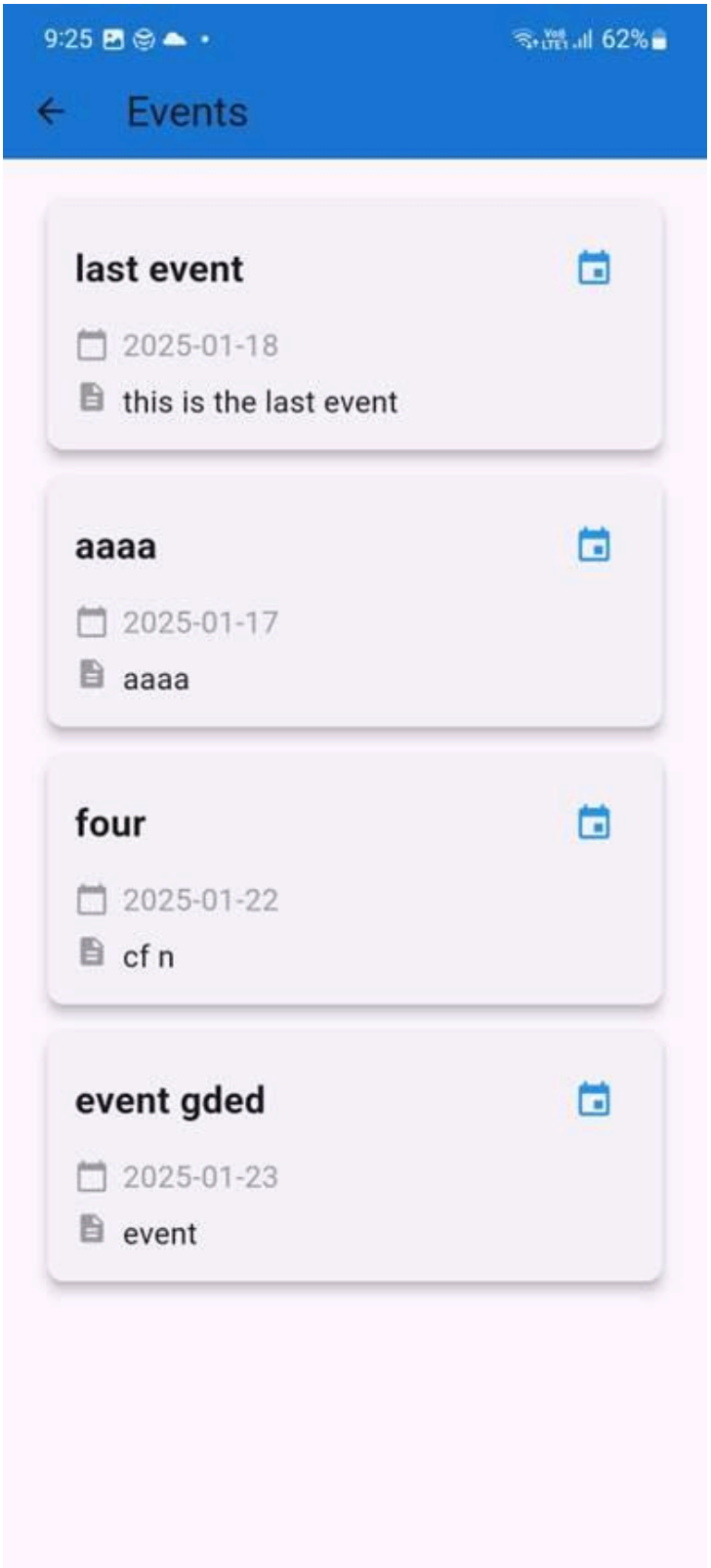
```dart
abstract class Command {
  Future<void> execute();
}
```


9:16
65%
Create Event
Event Name
Description
Date
Expected Attendance
Add Task
Create Event
Tasks for this Event


9:16
65%
Edit Events
last event
2025-01-18
this is the last event
Attendance: 6
Spring Festival
2025-01-28
children play and have fun in a fresh air
Attendance: 20
sports day
2025-01-30
different sports and entertainment facilities will be available
Attendance: 20
aaaa
2025-01-17
aaaa
Attendance: 50
Talent Show

Proxy pattern:

```dart
class EventsProxy {
  final FirestoreDatabaseService _firestoreDatabaseService = FirestoreDatabaseService();

  // Proxy method to fetch events based on user type
  Future<List<Event>?> fetchEvents(String userType) async {
    // If the user is not an Admin, apply filtering for events happening this week
    if (userType != 'Admin') {
      return await _fetchEventsForUser();
    } else {
      // For Admin, fetch all events
      return await _firestoreDatabaseService.fetchAllEvents();
    }
  }
  // Method for non-admin users, applying the week filter
  Future<List<Event>?> _fetchEventsForUser() async {
    try {
      // Fetch all events from Firestore
      final events = await _firestoreDatabaseService.fetchAllEvents();
      return _filterEventsForThisWeek(events!); // Filter to show only events happening this week
    } catch (e) {
      print('Error fetching events for user: $e');
      return [];
    }
  }

  // Filter events to only include those happening this week
  List<Event> _filterEventsForThisWeek(List<Event> events) {
    print(events);
    final now = DateTime.now();
    final startOfWeek = now.subtract(Duration(days: 1)); // Start of this week (Monday)
    final endOfWeek = startOfWeek.add(Duration(days: 7)); // End of this week (Sunday)
    List<Event> filtered = events.where((event) {
      DateTime eventDate = DateTime.parse(event.date); // Assuming event.date is in a parsable format
      return eventDate.isAfter(startOfWeek) && eventDate.isBefore(endOfWeek);
    }).toList();
    print(filtered);
    return filtered;
  }
}
```


9:25
62%
Events
last event
2025-01-18
this is the last event
aaaa
2025-01-17
aaaa
four
2025-01-22
cf n
event gded
2025-01-23
event

Adapter pattern :

```dart
import 'package:hope_home/models/Donation/donation.dart';

class DonationAdapter {
  Donation donation ;
  DonationAdapter(this.donation);
  Map<String,dynamic> ToFireStore(){

    return {
      'id': donation.id,
      'donorName': donation.donorName,
      'donorEmail': donation.donorEmail,
      'amount': donation.amount,
      'method': donation.method,
      'date': donation.date,
    };


  }
}
```

```dart
class EventAdapter{
  Event event;
  EventAdapter(this.event);
  Map<String,dynamic> ToFireStore(){

    return {
      'name': event.name,
      'description': event.description,
      'date': event.date,
      'attendance': event.attendance,
    };

  }
}
```

Builder design pattern:

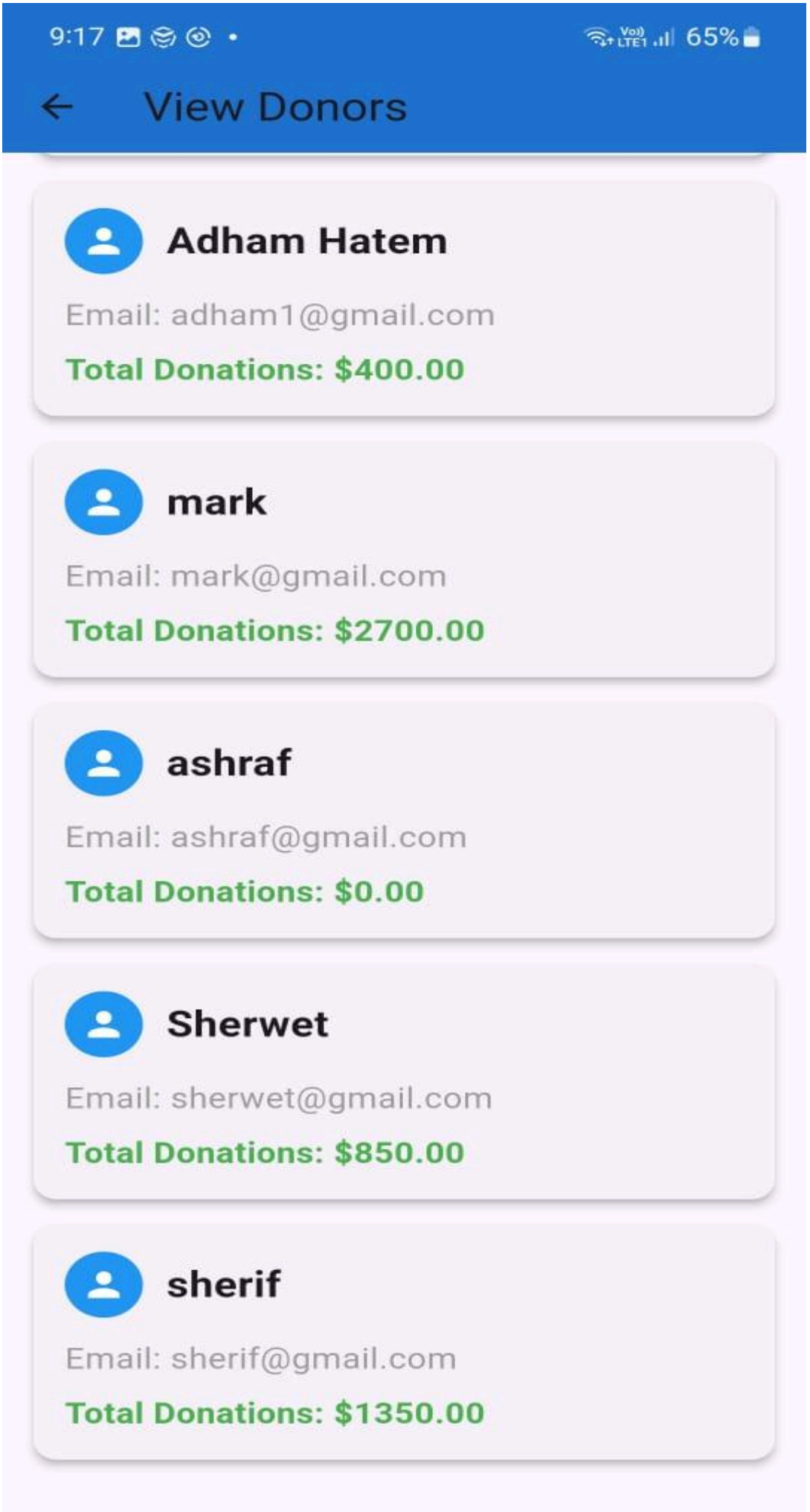
```dart
class EventBuilder {
  String? _id;
  String? _name;
  int? _attendance;
  String? _description;
  String? _date;

  EventBuilder setId(String id) {
    _id = id;
    return this;
  }

  EventBuilder setName(String name) {
    _name = name;
    return this;
  }

  EventBuilder setAttendance(int attendance) {
    _attendance = attendance;
    return this;
  }

  EventBuilder setDescription(String description) {
    _description = description;
    return this;
  }

  EventBuilder setDate(String date) {
    _date = date;
    return this;
  }

  Event build() {
    return Event(
      id: _id!,
      name: _name!,
      attendance: _attendance!,
      description: _description!,
      date: _date!,
    );
  }
}
```

Iterator pattern :

```dart
abstract class Iterator {
  bool hasNext();
  dynamic next();
}
```

```dart
abstract class EventIterator {
  bool hasNext();
  Event next();
}

class EventListIterator implements EventIterator {
  final List<Event> _events;
  int _currentIndex = 0;

  EventListIterator(this._events);

  @override
  bool hasNext() {
    return _currentIndex < _events.length;
  }

  @override
  Event next() {
    if (!hasNext()) {
      throw Exception("No more events.");
    }
    return _events[_currentIndex++];
  }
}
```

```dart
abstract class EventCollection {
  EventIterator createIterator();
}

class EventList implements EventCollection {
  final List<Event> _events = [];

  void addEvent(Event event) {
    _events.add(event);
  }

  List<Event> get events => _events;

  @override
  EventIterator createIterator() {
    return EventListIterator(_events);
  }
}
```


9:17
65%
View Donors
Adham Hatem
Email: adham1@gmail.com
Total Donations: $400.00
mark
Email: mark@gmail.com
Total Donations: $2700.00
ashraf
Email: ashraf@gmail.com
Total Donations: $0.00
Sherwet
Email: sherwet@gmail.com
Total Donations: $850.00
sherif
Email: sherif@gmail.com
Total Donations: $1350.00

```
class DonorIterator implements Iterator {
  final List<Donor> _donors;
  int _currentIndex = 0;

  DonorIterator(this._donors);

  @override
  bool hasNext() {
    return _currentIndex < _donors.length;
  }

  @override
  Donor next() {
    if (!hasNext()) throw Exception("No more donors");
    return _donors[_currentIndex++];
  }

  @override
  // TODO: implement current
  get current => throw UnimplementedError();

  @override
  bool moveNext() {
    // TODO: implement moveNext
    throw UnimplementedError();
  }
}
```

Strategy pattern :

```dart
abstract class CommunicationStrategy {
  Future<void> sendMessage(String recipient, String message, String type);
}
```

```dart
class EmailStrategy implements CommunicationStrategy {
  @override
  Future<void> sendMessage(String recipient, String message, String type) async {
    await FirebaseFirestore.instance.collection('messages').add({
      'recipient': recipient,
      'message': message,
      'type': type,
      'timestamp': FieldValue.serverTimestamp(),
    });
    print('Message stored as Email for $recipient');
  }
}
```

```dart
class SmsStrategy implements CommunicationStrategy {
  @override
  Future<void> sendMessage(String recipient, String message, String type) async {
    await FirebaseFirestore.instance.collection('messages').add({
      'recipient': recipient,
      'message': message,
      'type': type,
      'timestamp': FieldValue.serverTimestamp(),
    });
    print('Message stored as SMS for $recipient');
  }
}
```

```dart
class PushNotificationStrategy implements CommunicationStrategy {
  @override
  Future<void> sendMessage(String recipient, String message, String type) async {
    await FirebaseFirestore.instance.collection('messages').add({
      'recipient': recipient,
      'message': message,
      'type': type,
      'timestamp': FieldValue.serverTimestamp(),
    });
    print('Message stored as Push Notification for $recipient');
  }
}
```


9:18
64%
Select Communication Type
Email
SMS
Push Notification

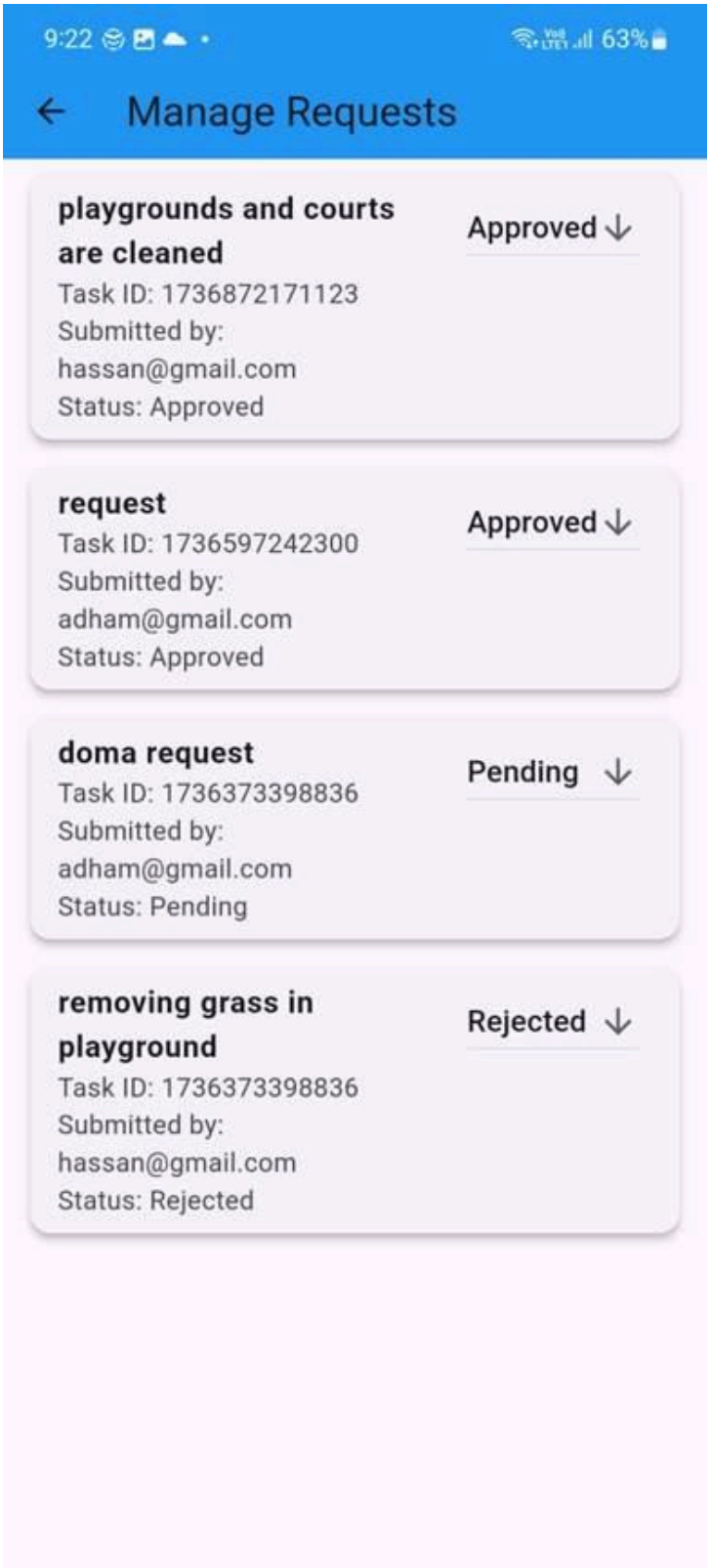
Observer pattern:

```dart
class UserProvider extends ChangeNotifier {
  // Singleton instance
  static final UserProvider _instance = UserProvider._internal();

  // Private constructor
  UserProvider._internal();

  // Public getter to access the singleton instance
  factory UserProvider() {
    return _instance;
  }

  // Current user
  myUser? _currentUser;

  myUser? get currentUser => _currentUser;

  // Set the current user and notify listeners
  void setUser(myUser? user) {
    _currentUser = user;
    notifyListeners();
  }

  void display() {
    print(_currentUser!.name + "\n" + _currentUser!.type);
  }

  // Optionally, clear the user data on logout
  void clearUser() {
    _currentUser = null;
    notifyListeners();
  }
}
```

State pattern:

```dart
abstract class RequestState {
  Future<void> handle(Request request);
  bool canEdit();
  String getStateName();
}
```

```dart
class ApprovedState implements RequestState {
  @override
  Future<void> handle(Request request) async {
    bool isConnected = await FirestoreDatabaseService().checkConnection();
    if (!isConnected) {
      return;
    }

    await FirestoreDatabaseService().notifyAdmin(request.id, "Request approved.");
  }

  @override
  bool canEdit() => false;

  @override
  String getStateName() => "Approved";
}

class RejectedState implements RequestState {
  @override
  Future<void> handle(Request request) async {
    bool isConnected = await FirestoreDatabaseService().checkConnection();
    if (!isConnected) {
      return;
    }
```


9:22
63%
Manage Requests
playgrounds and courts are cleaned
Approved
Task ID: 1736872171123
Submitted by:
hassan@gmail.com
Status: Approved
request
Approved
Task ID: 1736597242300
Submitted by:
adham@gmail.com
Status: Approved
doma request
Pending
Task ID: 1736373398836
Submitted by:
adham@gmail.com
Status: Pending
removing grass in playground
Rejected
Task ID: 1736373398836
Submitted by:
hassan@gmail.com
Status: Rejected

Façade pattern:

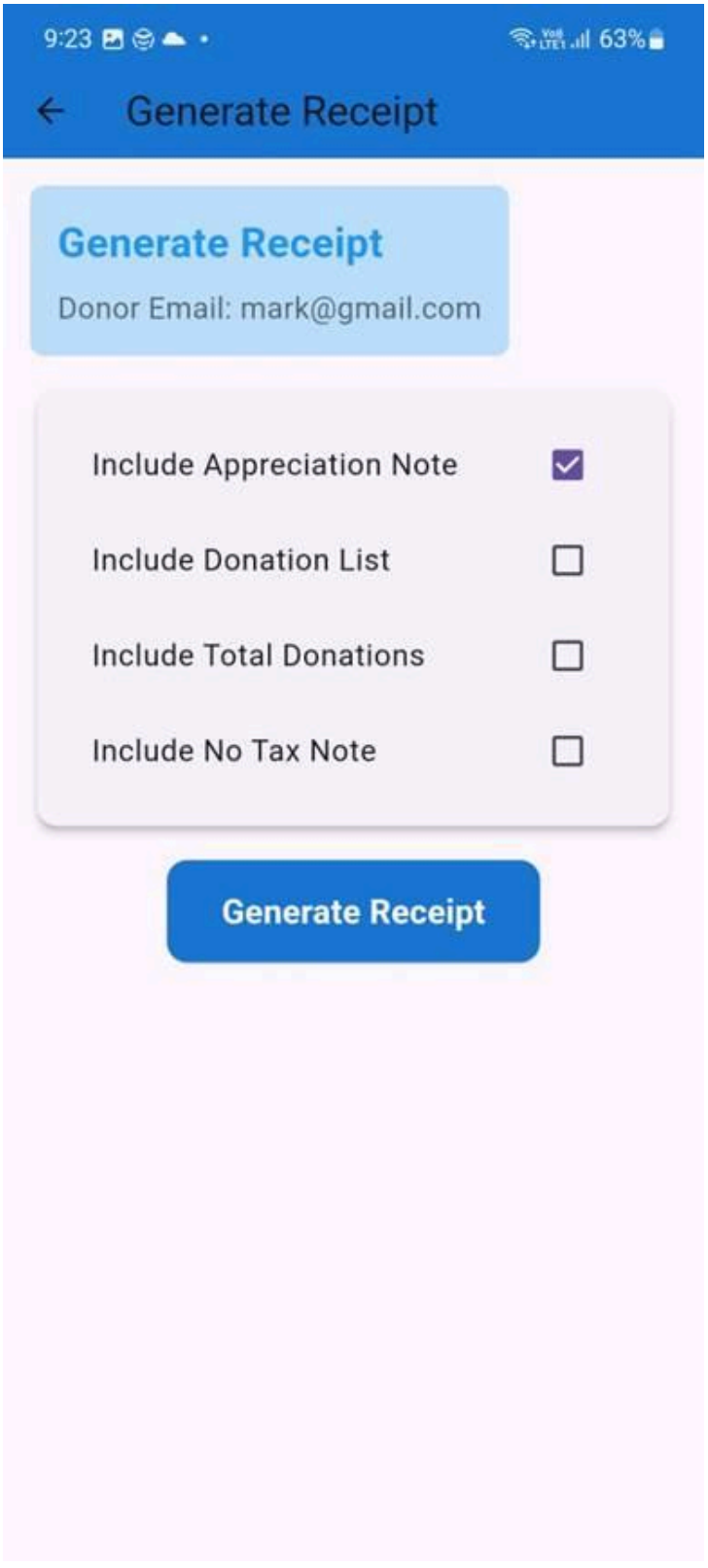
```dart
class UserServiceHelper {
  final AuthService authService;
  final DatabaseService databaseService;  // This will now refer to the interface

  UserServiceHelper({
    required this.authService,
    required this.databaseService,
  });

  // Login method
  Future<String?> loginWithEmailPassword(String email, String password) async {
    return await authService.login(email, password);
  }

  // Signup method
  Future<String?> signupWithEmailPassword(String email, String password, String name, String type) async {
    String? userId = await authService.signup(email, password, name, type);
    if (userId != null) {
      await databaseService.insertUser(userId, name, email, type);
    }
    return userId;
  }

  // Fetch user data
  Future<Map<String, dynamic>?> fetchUserData(String id) async {
    return await databaseService.fetchUserData(id);
  }
}
```

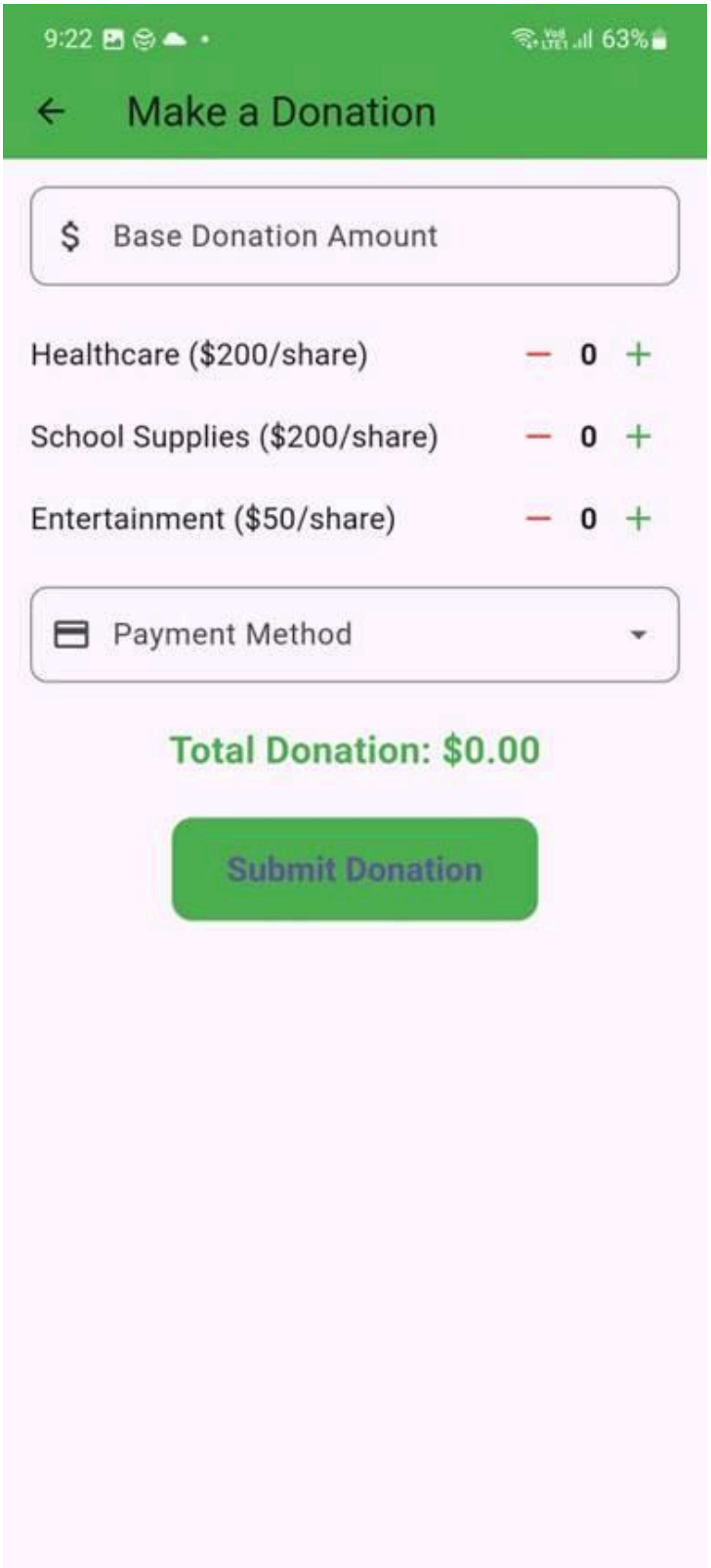
```dart
// ignore: empty_constructor_bodies
static UserLoginMailController createInstance(){
  return UserLoginMailController(
    facade: UserServiceHelper(
      authService: FirebaseAuthService(),
      databaseService: FirestoreDatabaseService(),
    ),  // UserServiceHelper
    userProvider: UserProvider(),
  );  // UserLoginMailController
}


@override
Future<String?> authenticate(String email, String password, [String? name, String? type]) async {
  try {
    String? userId = await facade.loginWithEmailPassword(email, password);

    if (userId != null) {
      var userData = await facade.fetchUserData(userId);
      if (userData != null) {
        String? userType = userData['type']?.toString(); // Force conversion to String

        if (userType == null || userType.isEmpty) {
          throw Exception('User type is missing or invalid.');
        }
```

Decorator pattern:

```dart
import 'donation_calculator.dart';

abstract class DonationDecorator implements DonationCalculator {
  final DonationCalculator donation;

  DonationDecorator(this.donation);

  @override
  double getCost() {
    return donation.getCost();
  }

  @override
  String getDescription() {
    return donation.getDescription();
  }
}
```

```dart
class DonationListDecorator extends ReceiptComponent {
  final ReceiptComponent receipt;
  final List<Donation> donations;

  DonationListDecorator(this.receipt, this.donations);

  @override
  String generateReceipt() {
    String donationDetails = donations
        .map((donation) =>
    "${donation.date}: ${donation.donorName} donated \$${donation.amount.toStringAsFixed(2)}")
        .join("\n");
    return "${receipt.generateReceipt()}\n\nDonation List:\n$donationDetails";
  }
}

class TotalDonationsDecorator extends ReceiptComponent {
  final ReceiptComponent receipt;
  final double total;

  TotalDonationsDecorator(this.receipt, this.total);

  @override
  String generateReceipt() {
    return "${receipt.generateReceipt()}\n\nTotal Donations: \$${total.toStringAsFixed(2)}";
  }
}
```


9:23
63%
Generate Receipt
Generate Receipt
Donor Email: mark@gmail.com
Include Appreciation Note
Include Donation List
Include Total Donations
Include No Tax Note
Generate Receipt


9:22
63%
Make a Donation
$ Base Donation Amount
Healthcare ($200/share) − 0 +
School Supplies ($200/share) − 0 +
Entertainment ($50/share) − 0 +
Payment Method
Total Donation: $0.00
Submit Donation

Singleton pattern:

```dart
class FirestoreDatabaseService implements DatabaseService {
  // Private static instance
  static final FirestoreDatabaseService _instance = FirestoreDatabaseService
      ._internal();

  // Factory constructor to return the single instance
  factory FirestoreDatabaseService() {
    return _instance;
  }

  // Private constructor
  FirestoreDatabaseService._internal();

  // Firestore instance
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;
```

Template pattern:

```dart
abstract class UserLoginTemplate {
    final UserServiceHelper facade;
    UserLoginTemplate({required this.facade});
    myUser? user;
    UserFactory? userFactory ;
  // The template method that outlines the steps
  Future<String?>  login(String email,String password,[String? name,String? type]) async {
    String? id = await authenticate(email,password,name??"",type??'');
    print(id);
    Map<String,dynamic> data = await fetchUserData(id) as Map<String,dynamic>;
    print(data);
    user = createUserObject(data);
    print(user);
    postLoginProcess(user);
    return data['type'];
  }

  // Step 1: Authentication (could be handled in Firebase)
  Future<String?> authenticate(String email,String password,[String? name,String? type]);

  // Step 2: Fetching user data from the DB (Firestore)

  Future<Map<String, dynamic>?> fetchUserData(id) async{
    return facade.fetchUserData(id);
  }




  myUser? createUserObject(Map<String,dynamic>? data) {
    userFactory = UserFactoryProducer.getFactory(data!['type']);
    user = userFactory!.createUser(data!);
    return user;
  }


  void postLoginProcess(myUser? user) {
  }
}
```