TheReminder Software Design Document

BİL 482 - Design Patterns Course Project

Course: BİL 482 - Design Patterns
Project: TheReminder - Accessible To-Do App with Reminder Modes
Team Members: Beyzanur Zeybek, Feyza Coşkun, Taha Mert Ağım, Özge Doğan, Talha Akbulut
Semester: 2024-2025 Spring
Instructor: Çiğdem Avcı

System Overview

TheReminder is an accessibility-first Flutter application designed to help users manage and track their daily tasks and reminders efficiently. This project serves as a comprehensive implementation of four fundamental design patterns studied in BİL 482 - Design Patterns course: Singleton, Observer, Decorator, and Strategy patterns.

Project Objectives

The primary objective of this project is to demonstrate practical application of design patterns in real-world software development scenarios. Each pattern was carefully selected to address specific architectural challenges in the task management domain:

1. **Singleton Pattern**: Ensures single instance management for database and notification services
2. **Observer Pattern**: Implements loose-coupled notification system for task reminders
3. **Decorator Pattern**: Provides dynamic accessibility features without modifying core classes
4. **Strategy Pattern**: Enables runtime selection of notification strategies

Academic Context

This project fulfills the requirements of BİL 482 Design Patterns course by demonstrating:

- Understanding of design pattern concepts and their practical applications
- Implementation of multiple patterns in a cohesive software system
- Analysis of pattern selection rationale and trade-offs
- Documentation of pattern usage with UML diagrams and code examples

System Context

The application operates as a standalone mobile application with the following external dependencies:

- **Flutter Framework**: Cross-platform UI framework
- **SQLite Database**: Local data persistence
- **Local Notifications**: System-level notification delivery
- **Device Settings**: Accessibility and system preferences

Design Pattern Integration Context

The system context demonstrates how design patterns interact with external systems:

- **Singleton Pattern** manages database connections and notification services
- **Observer Pattern** handles system-level notification delivery
- **Strategy Pattern** adapts to different device capabilities and user preferences
- **Decorator Pattern** responds to system accessibility settings

Key Features and Functionality

- Create, edit, and delete reminders
- Track tasks by time and completion status
- Priority-based task classification (High, Medium, Low)
- Scheduled local notifications
- Accessibility features (font size, contrast settings)
- Offline-first architecture with SQLite database
- Dynamic UI customization through decorators
- Multiple notification strategies (Audio, Visual, Vibration)

Design Pattern-Driven Features

Each feature is implemented using appropriate design patterns:

- **Task Management**: Singleton pattern for database operations
- **Notification System**: Observer pattern for event-driven notifications
- **Accessibility**: Decorator pattern for dynamic UI customization
- **Notification Types**: Strategy pattern for different notification behaviors

Assumptions and Dependencies

- Flutter SDK 3.8.1 or higher
- SQLite for local data storage

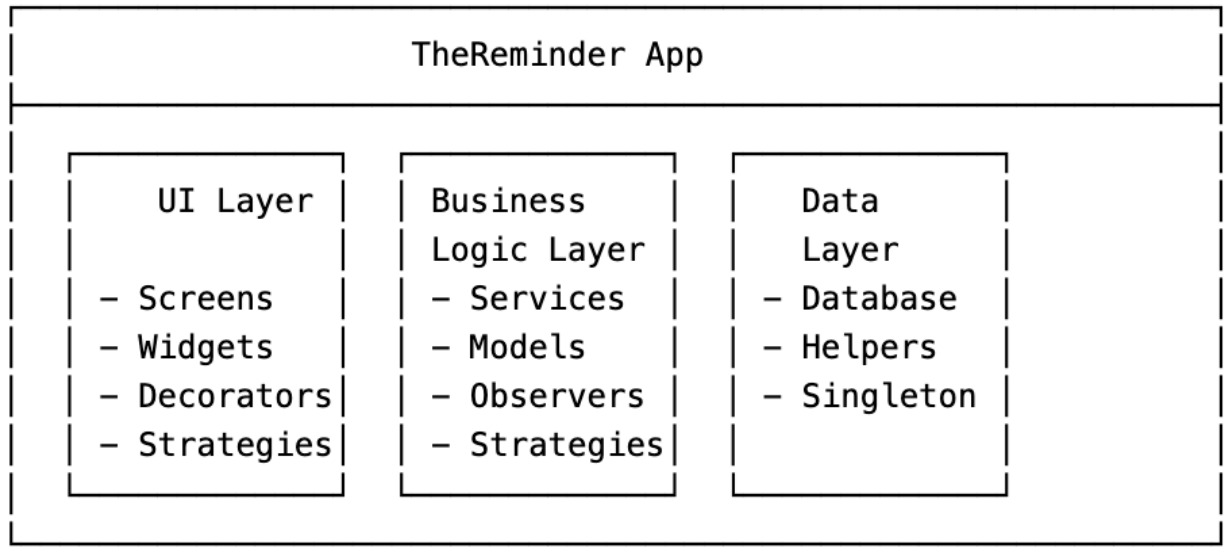
- Flutter Local Notifications for push notifications
- Shared Preferences for settings storage
- Timezone support for accurate scheduling
- Device notification permissions

Pattern-Specific Assumptions

- **Singleton**: Assumes single-threaded access to shared resources
- **Observer**: Assumes loose coupling between notification sources and handlers
- **Decorator**: Assumes runtime flexibility for UI customization
- **Strategy**: Assumes interchangeable notification behaviors

Architectural Design

System Architecture Diagram (High-Level)



Architectural Patterns and Styles

The application follows a **Layered Architecture** pattern with clear separation of concerns:

- **Presentation Layer**: Flutter widgets and screens
- **Business Logic Layer**: Services, observers, and models
- **Data Access Layer**: Database helpers and storage managers

Rationale for Architectural Decisions

1. **Layered Architecture**: Promotes separation of concerns and maintainability

2. **Offline-First**: Ensures functionality without internet connectivity
3. **Accessibility-First**: Prioritizes inclusive design from the start
4. **Singleton Pattern**: Ensures single database connection and resource optimization
5. **Observer Pattern**: Enables loose coupling for notification system
6. **Decorator Pattern**: Allows dynamic UI customization without modifying core classes
7. **Strategy Pattern**: Enables runtime selection of notification strategies

Design Pattern Selection Rationale

Why Singleton Pattern?

The Singleton pattern was chosen for database and notification services because:

- **Resource Management**: Database connections are expensive to create and maintain
- **State Consistency**: Ensures all parts of the application access the same data state
- **Memory Efficiency**: Prevents multiple instances from consuming unnecessary memory
- **Thread Safety**: Provides controlled access to shared resources

Why Observer Pattern?

The Observer pattern was selected for the notification system because:

- **Loose Coupling**: Notification sources don't need to know about specific handlers
- **Extensibility**: New notification types can be added without modifying existing code
- **Event-Driven Architecture**: Supports asynchronous notification delivery
- **Separation of Concerns**: Separates notification logic from task management logic

Why Decorator Pattern?

The Decorator pattern was implemented for accessibility features because:

- **Open-Closed Principle**: New accessibility features can be added without modifying existing code
- **Runtime Flexibility**: UI customization can be applied dynamically
- **Composition Over Inheritance**: Avoids complex inheritance hierarchies

- **Single Responsibility**: Each decorator handles one specific accessibility feature

Why Strategy Pattern?

The Strategy pattern was chosen for notification strategies because:

- **Runtime Selection**: Users can choose notification types at runtime
- **Extensibility**: New notification strategies can be easily added
- **Algorithm Encapsulation**: Each notification type is encapsulated in its own class
- **Elimination of Conditionals**: Replaces complex if-else statements with polymorphic behavior

Component Design

Subsystems and Modules

1. UI Layer (Presentation)

- **Screens**: Home, Create Task, Edit Task, Settings
- **Widgets**: Accessibility decorators, priority indicators
- **Navigation**: Flutter Navigator

2. Business Logic Layer

- **Services**: NotificationService, TaskService
- **Observers**: NotificationObserver
- **Strategies**: NotificationStrategy implementations
- **Models**: Task, Reminder, Priority, NotificationType

3. Data Layer

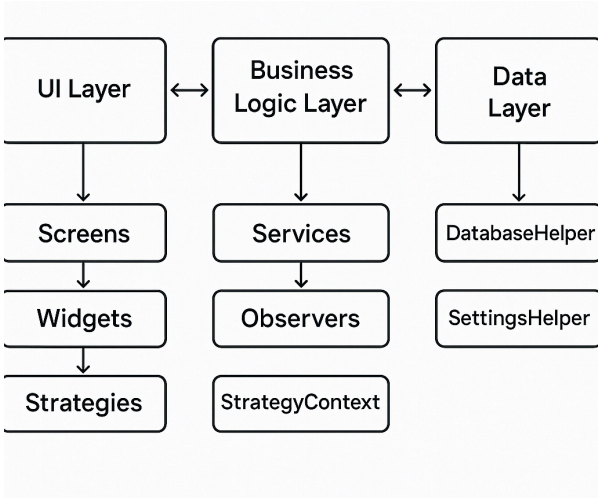
- **Database**: SQLite with DatabaseHelper
- **Storage**: SettingsHelper for preferences
- **Caching**: In-memory task list

Responsibilities of Each Component

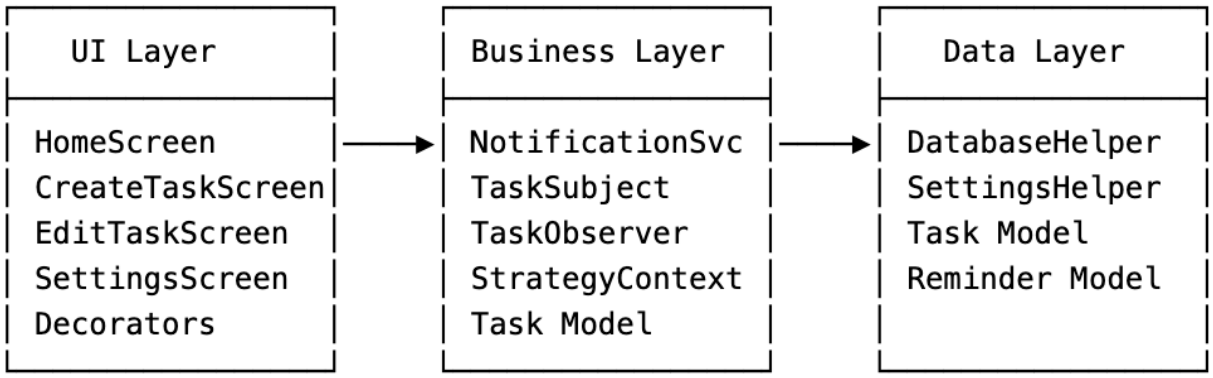
Component	Responsibility	Design Pattern Used
HomeScreen	Display task list, handle task interactions	UI Component
CreateTaskScreen	Task creation with validation	UI Component
EditTaskScreen	Task modification and deletion	UI Component

DatabaseHelper	SQLite operations, singleton data access	Singleton
NotificationService	Central notification management	Singleton
TaskSubject	Observer pattern subject for task scheduling	Observer
AccessibilityDecorator	Dynamic UI accessibility features	Decorator
Task	Data model for task information	Data Model
Reminder	Data model for reminder settings	Data Model
NotificationStrategy	Strategy pattern for notification types	Strategy
NotificationStrategyContext	Manages strategy selection and execution	Strategy

Interfaces Between Components

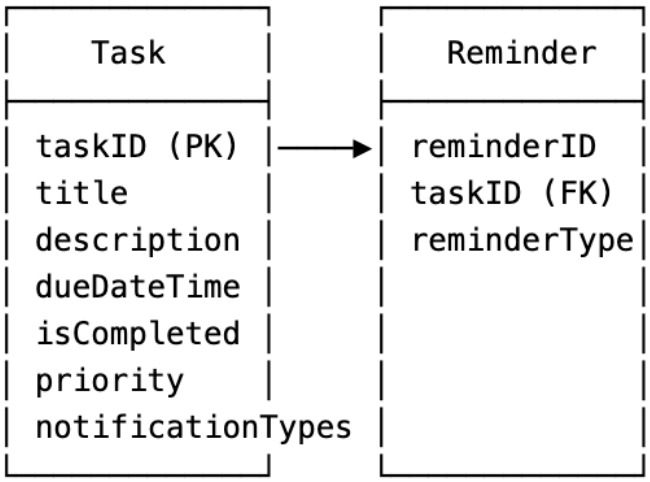


Component Diagrams



Data Design

Data Model / ER Diagram



Data Storage (Database or File Structure)

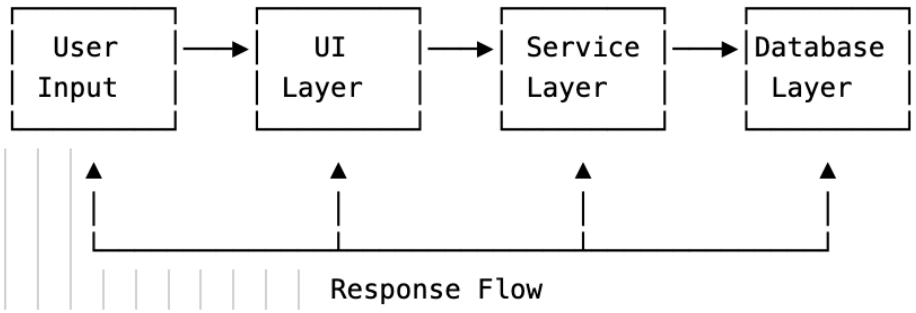
Task Table

```
CREATE TABLE Task (
  taskID INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  description TEXT,
  dueDateTime TEXT NOT NULL,
  isCompleted INTEGER DEFAULT 0,
  priority TEXT,
  notificationTypes TEXT DEFAULT 'Visual'
);
```

Reminder Table

```
CREATE TABLE Reminder (
  reminderID INTEGER PRIMARY KEY AUTOINCREMENT,
  taskID INTEGER NOT NULL,
  reminderType TEXT,
  FOREIGN KEY (taskID) REFERENCES Task(taskID)
);
```

Data Flow Diagrams



Data Validation Rules

- Task title cannot be empty
- Due date must be a valid DateTime
- Priority must be one of: High, Medium, Low
- Task ID must be unique
- Reminder taskID must reference existing Task
- NotificationTypes must include at least Visual type

Design Patterns

Applied Design Patterns

1. Singleton Pattern

Context and Justification: Multiple reminder behaviors (data storage, notification management) must have a single point of access to ensure consistency across the application. This pattern addresses the need for centralized resource management in a multi-component system.

Problem Solved:

- Multiple database connections causing resource waste
- Inconsistent state across different parts of the application
- Uncontrolled access to shared resources

Solution: Ensures that a class has only one instance and provides a global point of access to that instance.

Application in Architecture:

- **DatabaseHelper**: Ensures single database connection and consistent data access
- **NotificationService**: Centralizes notification management across the app

Implementation:

```
class DatabaseHelper {
  static final DatabaseHelper instance = DatabaseHelper._();
  DatabaseHelper._();

  static Database? _db;
  Future<Database> get database async {
        _db ??= await getDatabase();
        return _db!;
  }
}
```

Benefits:

- Resource optimization
- Thread-safe access
- Consistent state management
- Controlled access to shared resources

Trade-offs:

- Global state can make testing difficult
- Potential for tight coupling
- Memory usage if not properly managed

2. Observer Pattern

Context and Justification: Reminder system needs to notify users at the correct time without tightly coupling with specific notification mechanisms. This pattern enables a one-to-many dependency relationship between objects.

Problem Solved:

- Tight coupling between task scheduling and notification delivery
- Difficulty in adding new notification types
- Synchronization issues between multiple notification handlers

Solution: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Application in Architecture:

- **TaskSubject**: Acts as the subject and manages task scheduling
- **NotificationObserver**: Handles phone notifications

Implementation:

```
abstract class Observer {
  void update(String message, Map<String, dynamic> data);
}

class TaskSubject implements Subject {
  final List<Observer> _observers = [];

  void scheduleTask(Task task) {
        // Schedule timer and notify observers when time comes
  }
}
```

Benefits:

- Loose coupling between components
- Extensible notification system
- Multiple notification types support
- Event-driven architecture

Trade-offs:

- Potential memory leaks if observers are not properly removed
- Order of notification delivery is not guaranteed
- Debugging can be complex with many observers

3. Decorator Pattern

Context and Justification: Task attributes like priority, font size, or color need to be added dynamically without modifying the base task class. This pattern allows for flexible composition of behaviors.

Problem Solved:

- Need to add new UI features without modifying existing code
- Complex inheritance hierarchies for UI customization
- Runtime flexibility for accessibility features

Solution: Attaches additional responsibilities to an object dynamically, providing a flexible alternative to subclassing for extending functionality.

Application in Architecture:

- **AccessibilityDecorator**: Base decorator for accessibility features
- **FontDecorator**: Adds font size scaling
- **ContrastDecorator**: Adds high contrast theme

- **PriorityDecorator**: Adds color coding for priorities

Implementation:

```
class AccessibilityDecorator extends StatelessWidget {
  final Widget child;
  const AccessibilityDecorator(this.child, {super.key});

  @override
  Widget build(BuildContext context) {
        return child;
  }
}

class FontDecorator extends AccessibilityDecorator {
  final double fontSize;

  @override
  Widget build(BuildContext context) {
        final t = TextScaler.linear(fontSize);
        return MediaQuery(
        data: MediaQuery.of(context).copyWith(textScaler: t),
        child: child
        );
  }
}
```

Benefits:

- Dynamic feature addition
- Runtime flexibility
- Maintains single responsibility principle
- Avoids complex inheritance hierarchies

Trade-offs:

- Can lead to many small classes
- Debugging can be complex with many decorators
- Performance overhead with multiple decorators

4. Strategy Pattern

Context and Justification: Multiple reminder behaviors (audio, visual, vibration) must be interchangeable without altering task logic. This pattern enables runtime selection of algorithms.

Problem Solved:

- Complex conditional logic for different notification types
- Difficulty in adding new notification strategies
- Tight coupling between task logic and notification behavior

Solution: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Application in Architecture:

- **NotificationStrategy**: Abstract strategy interface
- **AudioStrategy**: Audio notification implementation
- **VisualStrategy**: Visual notification implementation
- **VibrationStrategy**: Vibration notification implementation
- **AudioVibrationStrategy**: Combined audio and vibration
- **NotificationStrategyContext**: Manages strategy selection and execution
- **NotificationStrategyFactory**: Creates appropriate strategies

Implementation:

```
abstract class NotificationStrategy {
  Future<void> execute(Map<String, dynamic> data, Color color);
  String get name;
}

class AudioStrategy implements NotificationStrategy {
  @override
  String get name => 'Audio';

  @override
  Future<void> execute(Map<String, dynamic> data, Color color) async {
        // Audio notification implementation
  }
}

class NotificationStrategyContext {
  NotificationStrategy? _strategy;

  void setStrategy(NotificationStrategy strategy) {
        _strategy = strategy;
  }

  Future<void> executeStrategy(Map<String, dynamic> data) async {
```

```
        if (_strategy != null) {
        await _strategy!.execute(data, Colors.blue);
        }
  }
}

class NotificationStrategyFactory {
  static NotificationStrategy createStrategy(List<String> types) {
        if (types.length == 3) {
        return AudioVibrationStrategy();
        } else if (types.length == 2) {
        switch (types[1].toLowerCase()) {
        case 'vibration':
        return VibrationStrategy();
        case 'audio':
        return AudioStrategy();
        default:
        return AudioVibrationStrategy();
        }
        } else {
        return VisualStrategy();
        }
  }
}
```

Benefits:

- Runtime strategy selection
- Easy to add new notification types
- Maintains single responsibility principle
- Factory pattern for strategy creation
- Eliminates complex conditional statements

Trade-offs:

- Increased number of classes
- Potential over-engineering for simple cases
- Learning curve for understanding strategy relationships

Pattern Interaction Analysis

How Patterns Work Together

1. **Singleton + Observer**: DatabaseHelper (Singleton) provides data to TaskSubject (Observer), which notifies observers

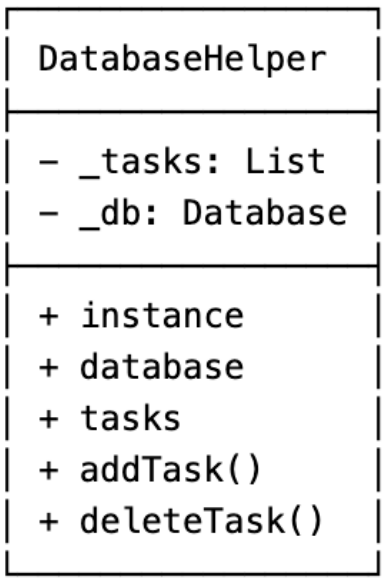
2. **Observer + Strategy**: TaskSubject (Observer) uses NotificationStrategy (Strategy) to determine notification behavior
3. **Decorator + Strategy**: UI decorators can be combined with notification strategies for comprehensive customization
4. **Singleton + Strategy**: NotificationService (Singleton) manages different strategies

Pattern Selection Criteria

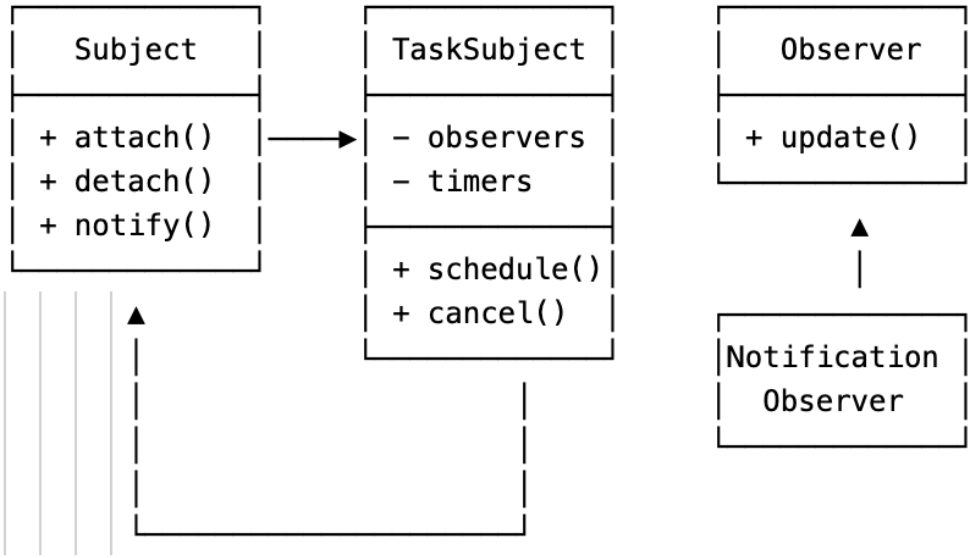
- **Singleton**: When exactly one instance is needed for resource management
- **Observer**: When loose coupling is required for event-driven systems
- **Decorator**: When runtime composition of behaviors is needed
- **Strategy**: When multiple algorithms need to be interchangeable

UML Diagrams

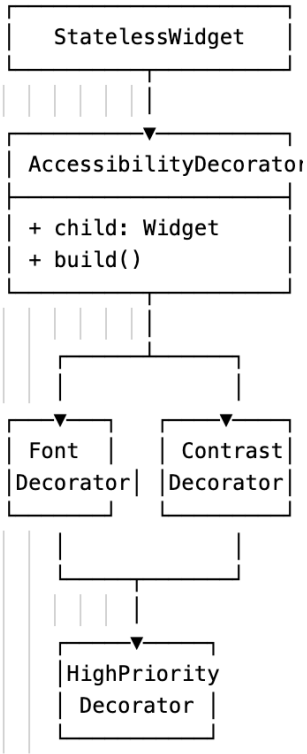
Singleton Pattern UML



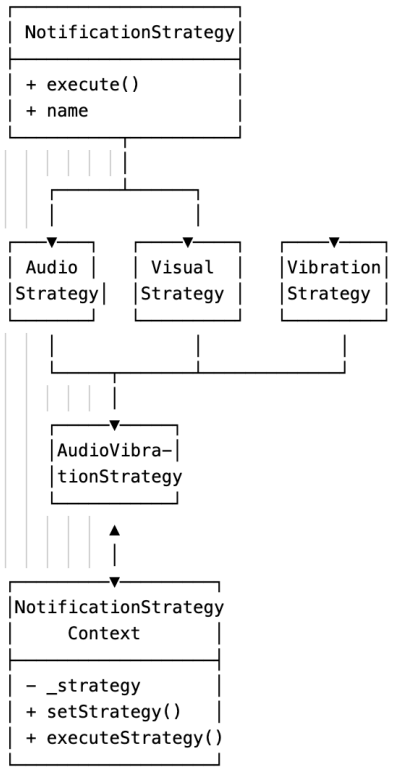
Observer Pattern UML



Decorator Pattern UML



Strategy Pattern UML



Implementation Notes

Technology Stack

- **Frontend**: Flutter 3.8.1
- **Database**: SQLite (sqflite package)
- **Notifications**: flutter_local_notifications
- **State Management**: Flutter's built-in StatefulWidget
- **Dependency Management**: pubspec.yaml

Key Implementation Decisions

1. **Singleton for Database**: Ensures single connection and thread safety
2. **Observer for Notifications**: Allows multiple notification types
3. **Decorator for Accessibility**: Enables runtime UI customization
4. **Strategy for Notification Types**: Enables runtime strategy selection
5. **SQLite for Storage**: Provides reliable offline data persistence

Code Organization

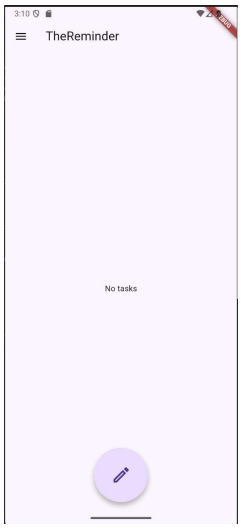
```
lib/
├── model/              # Data models
├── screens/            # UI screens
```

```
├── services/           # Business logic services
├── observer/           # Observer pattern implementation
│   ├── notification_strategy.dart  # Strategy pattern
│   └── notification_observer.dart  # Observer implementation
├── widgets/            # Reusable UI components
├── db/         # Data access layer
└── main.dart           # Application entry point
```

Design Pattern Implementation Guidelines

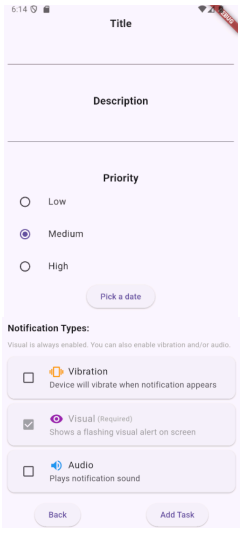
1. **Follow SOLID Principles**: Each pattern implementation adheres to SOLID principles
2. **Documentation**: Comprehensive comments explaining pattern usage
3. **Testing**: Unit tests for each pattern implementation
4. **Error Handling**: Proper exception handling in pattern implementations

User Interface Design



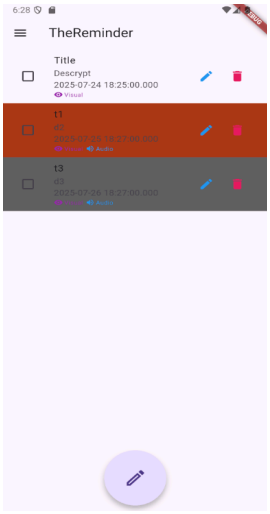
Initial Home Screen (Case of having no tasks)

- If there's no task planned or written, a centered message displayed "No Tasks"
- The FAB (Floating Action Button) is at the bottom of the screen. It's used to add a new task
- On the top bar, there's a title named TheReminder and the menu bar on the left which includes Settings. In Settings part, text size can be set (small, medium or large) and also contrast(dark) mode can be enabled



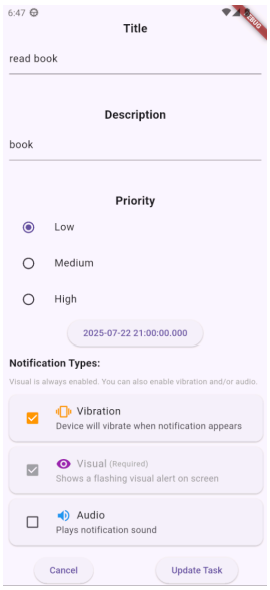
Create A New Task

- Users can input Title, Description, Priority and DueDateTime. If the user leaves those features blank, it throws a warning to fill the blanks
- Users can set the priority of the task (Low,Medium,High)
- Users can decide which notification type to send when the date comes to notify (Vibration, Visual (Stable) and Audio)
- Two buttons to go back to the home screen and to add the task, inputted to database (when updating, there're buttons of cancel and update in following. Other parts remain same with this screen)



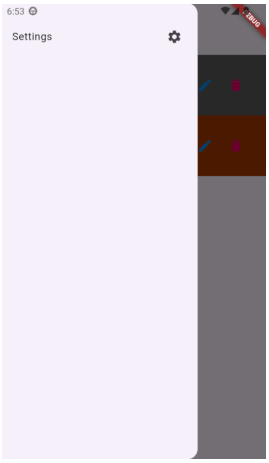
Home Screen with Tasks

- Tasks are colored depending on priorities
    - Black Red means high priority
    - Gray means low priority
    - No color means medium priority
- Blue mark is to edit/update the task (Confirmed that it updates)
- Red Trash icon is to delete task (Confirmed that it deletes)
- Checkbox is to mark the task as it's completed
- Details of the task are shown which are title, description, due date time and notification types used to notify



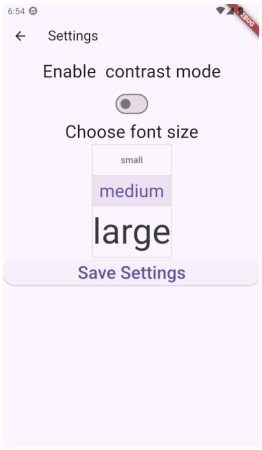
Edit Existing Tasks

- Users can edit the Title, Description, Priority and DueDateTime of an existing task. If the user leaves those fields blank, a warning is shown to prompt the user to fill them in.
- Users can change the priority of the task (Low, Medium, High).
- Users can update the notification type that will trigger when the due date arrives (Vibration, Visual (Stable), and Audio).
- Two buttons are provided to either go back to the home screen or to update the task, which updates the modified task in the database.



Navigation Drawer Screen

Screen that provides access to the settings button.



Settings Screen

- Users can enable or disable contrast mode using the toggle switch. This is intended to improve visual accessibility for users who need higher contrast.

- Users can select a preferred font size from the options: Small, Medium, or Large. The currently selected size is visually emphasized.

- A Save Settings button is provided to apply and store the user's visual preferences.

UI Mockups or Wireframes

The application follows Material Design principles with accessibility considerations:

1. **Home Screen**: List of tasks with priority indicators
2. **Create Task Screen**: Form with validation and accessibility options
3. **Edit Task Screen**: Pre-filled form with update capabilities
4. **Settings Screen**: Accessibility and notification preferences

Accessibility Features

- **Font Scaling**: Dynamic text size adjustment
- **High Contrast**: Dark theme for better visibility
- **Priority Colors**: Visual indicators for task importance
- **Screen Reader Support**: Semantic labels and descriptions

Design Pattern Integration in UI

- **Decorator Pattern**: Applied to UI components for accessibility
- **Strategy Pattern**: Used for notification type selection in settings
- **Observer Pattern**: Handles UI updates when notifications are triggered

External Interfaces

APIs

- **Local Notifications API**: System-level notification delivery
- **SQLite API**: Local data persistence
- **Shared Preferences API**: Settings storage

Third-party Systems

- **Flutter Framework**: Cross-platform development
- **Dart Language**: Programming language
- **Android/iOS Platforms**: Native platform integration

Pattern Integration with External Systems

- **Singleton**: Manages connections to external APIs
- **Observer**: Responds to system-level events
- **Strategy**: Adapts to different platform capabilities
- **Decorator**: Responds to system accessibility settings

Performance Considerations

Performance Requirements

The application does not have specific performance requirements as it is primarily a learning project for design patterns. However, general usability considerations include:

- **App Launch Time**: Should be reasonable for a mobile application
- **Task Creation**: Should feel responsive to user input
- **Notification Delivery**: Should work reliably within system constraints
- **Database Operations**: Should not cause noticeable delays

Scalability & Optimization Strategies

1. **Lazy Loading**: Load tasks on demand
2. **Caching**: In-memory task list
3. **Efficient Queries**: Optimized SQLite queries
4. **Background Processing**: Async notification scheduling

Pattern-Specific Performance Considerations

- **Singleton**: Memory usage optimization through lazy initialization
- **Observer**: Memory leak prevention through proper observer removal
- **Decorator**: Performance impact of multiple decorator layers
- **Strategy**: Runtime overhead of strategy selection

Error Handling and Logging

Exception Management

```
try {
  await db.addTask(task);
```

```
} catch (e) {
  log('Error adding task: $e');
  // Show user-friendly error message
}
```

Logging Mechanisms

- **Dart's built-in logging**: dart:developer package
- **Error tracking**: Structured error logging
- **Performance monitoring**: Operation timing logs

Pattern-Specific Error Handling

- **Singleton**: Exception handling for resource initialization
- **Observer**: Error handling in notification delivery
- **Decorator**: Graceful degradation when decorators fail
- **Strategy**: Fallback strategies when primary strategy fails

Design for Testability

Testing Approach

Manual testing was conducted throughout the development process to ensure functionality:

- **Unit Testing**: Basic functionality testing of individual components
- **Integration Testing**: Testing pattern interactions and data flow
- **UI Testing**: Manual testing of user interface components
- **Pattern Testing**: Verification of design pattern implementations

Testing Results

All design patterns were manually tested and verified to work correctly:

- **Singleton Pattern**: Confirmed single instance behavior
- **Observer Pattern**: Verified notification delivery system
- **Decorator Pattern**: Tested accessibility feature composition
- **Strategy Pattern**: Validated notification strategy selection and execution

Change Log

Version 1.1.0 (Current)

- Singleton Pattern: DatabaseHelper and NotificationService (June 2025)
- Observer Pattern: TaskSubject and notification observers (July 2025)

- Decorator Pattern: Accessibility decorators (July 2025)
- Strategy Pattern: NotificationStrategy implementations (July 2025)

Academic Milestones

- **June 2025**: Singleton Pattern implementation and documentation
- **July 2025**: Observer, Decorator, and Strategy Pattern implementation and testing

Planned Features

- Additional accessibility features
- Cloud synchronization
- Multi-language support

Future Work / Open Issues

Additional Features

1. **Cloud Sync**: Backup and restore functionality
2. **Categories**: Task categorization system
3. **Recurring Tasks**: Repeat task functionality
4. **Statistics**: Task completion analytics

Technical Debt

1. **Code Documentation**: Improve inline documentation
2. **Error Handling**: More comprehensive error scenarios
3. **Performance**: Optimize database queries
4. **Testing**: Increase test coverage

Academic Improvements

1. **Pattern Documentation**: More detailed pattern usage examples
2. **Performance Analysis**: Benchmark pattern implementations
3. **Alternative Patterns**: Consider other patterns for specific use cases
4. **Pattern Combinations**: Explore more complex pattern interactions

Conclusion

TheReminder application successfully demonstrates the practical application of four fundamental design patterns studied in BİL 482 Design Patterns course. Each pattern was carefully selected and implemented to address specific architectural challenges:

1. **Singleton Pattern**: Ensures efficient resource management and consistent data access
2. **Observer Pattern**: Enables flexible notification system with loose coupling
3. **Decorator Pattern**: Provides dynamic accessibility features without modifying core classes
4. **Strategy Pattern**: Enables runtime selection of notification strategies

These patterns work together to create a maintainable, extensible, and user-friendly task management application that prioritizes accessibility and offline functionality. The implementation demonstrates understanding of pattern selection rationale, trade-offs, and practical application in real-world scenarios.

Learning Outcomes

- Understanding of when and how to apply design patterns
- Practical experience with pattern implementation in Flutter/Dart
- Analysis of pattern interactions and dependencies
- Documentation and presentation of pattern usage
- Testing and validation of pattern implementations

This project serves as a comprehensive example of how design patterns can be effectively combined to create robust, maintainable software systems.