

German Learning App

Technical Architecture and System Design

Nico Abramowski

December 1, 2025

Abstract

In this report, the design and development of the Learning App in German, an Android application optimized for foreign language vocabulary learning via algorithmic reinforcement, are described. In addition to addressing foreign language learning and retention issues in memory, the application incorporates a Spaced Repetition System (SRS) optimized via an SM-2 algorithm. The app is developed on top of an offline-capable infrastructure stack that incorporates Kotlin and Jetpack Compute. It follows the principles of Clean Architecture and the Model-View-ViewModel structure to maintain strong separation of concerns and ease of testing and maintenance. Data storage is taken care of by the Room Persistence Library, which abstracts SQLite functionality and implements it in an optimal manner as part of the local storage system. In this document, the modular nature of this system, including domain modeling of linguistic units, repository pattern, and the use of reactivity for stateful systems in computing user views, will be discussed.

Contents

1	Introduction	4
1.1	Context and Motivation	4
1.2	Project Objectives	4
1.3	Scope of the Application	4
2	System Architecture	4
2.1	Architectural Overview	5
2.2	High-Level Diagram	5
2.3	Component Interaction	6
2.3.1	Presentation Layer	6
2.3.2	Domain Layer	6
2.3.3	Data Layer	6
3	Domain Layer	6
3.1	Core Domain Models	6
3.2	Repository Interfaces	7
3.3	Use Cases (Interactors)	7
3.3.1	GetNextCardUseCase	7
3.3.2	RateCardUseCase	7
3.4	Spaced Repetition Scheduler	7
4	Data Layer	8
4.1	Persistence Strategy	8
4.2	Repository Implementation	8
4.3	Database Schema (Entity Models)	8
4.3.1	CardEntity (cards)	8
4.3.2	ReviewStateEntity (review_states)	8
4.3.3	DeckEntity (decks)	9
4.4	Data Access Objects (DAOs)	9
5	Spaced Repetition Logic	9
5.1	Algorithm Design	9
6	Core Algorithms and Data Flow	9
6.1	Dashboard Initialization	9
6.2	Spaced Repetition Scheduler (SRS)	10
6.3	Study Session Orchestration	10
6.4	Review Submission Workflow	11
6.5	Deck Retrieval Strategy	11
7	User Flow and Interaction	12
7.1	User Journey	12
7.2	Flowchart Visualization	13
7.3	Interaction Design	13
8	User Interface (UI)	13
8.1	Navigation Structure	14

1 Introduction

1.1 Context and Motivation

In this modern world of e-learning, Mobile Assisted Language Learning (MALL) systems are recognized as the driving forces behind independent language development. Learning apps for any language, including the German language, are abundant in numbers. However, some of them lack an effective and science-supported step for locking in new words and grammatical rules in memory for the long run or are too cluttered and difficult to navigate in terms of layout and functionality. Here comes the need for the “German Learning App” that aims to provide learners with just that: an optimal and comfortable platform for mastering the core vocabulary and grammar of the language in question, taking into account specific features of the language such as case declensions and sentence construction.

1.2 Project Objectives

The primary objectives of this development project are the following:

1. **Algorithmic Efficiency:** To implement a custom scheduling algorithm based on SM-2 principles that dynamically adjusts review intervals based on user performance (Again, Good, Easy), thereby optimizing the forgetting curve.
2. **Architectural Robustness:** To engineer a scalable Android application using the industry-standard Clean Architecture pattern. This ensures that business logic (SRS algorithms) remains decoupled from the UI and data frameworks, facilitating future expansion into features like AI-voice integration or cloud synchronization.
3. **User-Centric Design:** To leverage Jetpack Compose for building a modern, reactive user interface that provides immediate feedback and seamless navigation, enhancing user engagement and daily habit formation.

1.3 Scope of the Application

The initial release of the application focuses on the following core functionalities:

- **Vocabulary Acquisition:** Flashcards for fundamental German nouns and verbs.
- **Grammar Practice:** Specialized card types for practicing German cases (Nominalive, Accusative, Dative) through gap-fill exercises.
- **Sentence Construction:** Scrambled sentence exercises to train syntax and word order.
- **Progress Tracking:** A dashboard visualizing daily review loads and new card intake to maintain user motivation.

2 System Architecture

The system architecture of the *German Learning App* adheres to the principles of Clean Architecture, ensuring a strict separation of concerns between the User Interface (UI),

Business Logic (Domain), and Data Persistence layers. This modular design facilitates testability, scalability, and maintainability.

2.1 Architectural Overview

The application is structured into three primary layers:

1. **UI Layer (Presentation):** Responsible for rendering the user interface using Jetpack Compose. It observes state changes from the ViewModels and reacts to user input.
2. **Domain Layer (Business Logic):** Contains the core business rules, including entities (e.g., Card, Deck) and use cases (e.g., GetNextCardUseCase). This layer is platform-agnostic and does not depend on the UI or Data layers.
3. **Data Layer (Persistence):** Manages data retrieval and storage. It implements the repository interfaces defined in the Domain layer, coordinating between local storage (Room Database) and potential remote sources.

2.2 High-Level Diagram

Figure 1 illustrates the interactions between the varying components of the system. The directional arrows indicate the flow of dependencies, adhering to the rule that inner layers (Domain) should never depend on outer layers (UI, Data).

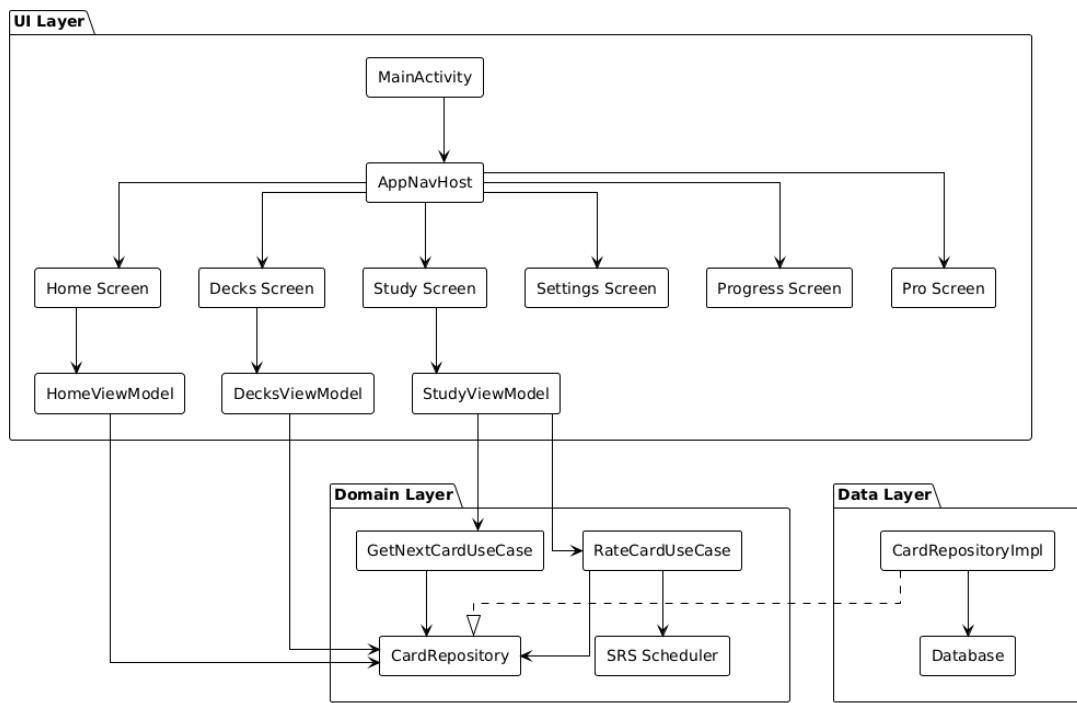


Figure 1: High-Level System Architecture Diagram showing the separation of UI, Domain, and Data layers.

2.3 Component Interaction

2.3.1 Presentation Layer

The UI is built using a single-activity architecture where `MainActivity` hosts an `AppNavHost`. This navigation host manages the composition of varying screens (`HomeScreen`, `StudyScreen`, etc.). Each screen corresponds to a specific `ViewModel` that exposes a reactive `UiState`.

2.3.2 Domain Layer

The business logic is encapsulated in Use Cases, which are single-purpose classes that orchestrate the flow of data. For example, the `RateCardUseCase` interacts with the SRS `Scheduler` to calculate the next review interval before persisting the state via the `CardRepository`.

2.3.3 Data Layer

The Data layer provides a concrete implementation of the `CardRepository`. It utilizes the Room persistence library to map Java/Kotlin objects to SQLite database tables. A `FakeCardRepository` is also utilized during development and testing to provide deterministic data without requiring a full database instance.

latex

3 Domain Layer

The Domain Layer represents the core of the application's architecture. It encapsulates the business rules, enterprise models, and data transformation logic that are specific to the German Learning App. In accordance with Clean Architecture principles, this layer is completely independent of the UI (Android framework) and Data (Database/Network) layers, ensuring high testability and stability.

3.1 Core Domain Models

The domain models are pure Kotlin data classes that represent the fundamental entities of the application. They are devoid of any framework-specific annotations (e.g., Room or Retrofit) to maintain separation of concerns.

- **Card:** The primary unit of learning. It encapsulates the content to be mastered (Front/Back text), its type (Vocabulary, Case, Sentence), and its association with a specific deck.
- **Deck:** Represents a collection of cards grouped by thematic or grammatical category (e.g., "Basic Vocabulary", "Accusative Case"). It includes metadata such as difficulty level and pro-status access control.
- **ReviewState:** A critical model for the SRS algorithm, this entity tracks the user's memory retention metrics for a specific card. It stores the `easeFactor`, `intervalDays`, and the calculated `dueDate`.

3.2 Repository Interfaces

To decouple the business logic from data retrieval, the Domain Layer defines Repository Interfaces. These interfaces act as a contract that the Data Layer must fulfill, adhering to the Dependency Inversion Principle.

The `CardRepository` interface defines the following key operations:

- `getDueCards(now: Long)`: Retrieves cards whose review date has passed.
- `getNewCards(limit: Int)`: Fetches a batch of unseen cards for the daily session.
- `saveReviewState(state: ReviewState)`: Persists the updated retention metrics after a user review.

3.3 Use Cases (Interactors)

Use Cases orchestrate the flow of data to and from the entities, implementing specific business rules. They serve as the entry point for the UI layer to interact with the domain.

3.3.1 GetNextCardUseCase

This use case determines the next optimal card to present to the user during a study session. It implements the logic for:

- Prioritizing due reviews to ensure retention.
- Interleaving new cards based on the daily limit configuration.
- Handling different study modes (e.g., "Review Only" vs. "Mixed Mode").

3.3.2 RateCardUseCase

Responsible for processing the user's feedback (Again, Good, Easy), this use case:

1. Retrieves the current review state of the card.
2. Invokes the SRS Scheduler to calculate the new interval and ease factor.
3. Persists the updated state via the repository, ensuring the card is rescheduled correctly.

3.4 Spaced Repetition Scheduler

The heart of the application is the SRS Scheduler, a domain service that implements the mathematical logic for memory retention. Based on a modified SM-2 algorithm, it dynamically adjusts the review schedule:

- **Interval Calculation:** Determines the number of days until the next review ($I_n = I_{n-1} \times EF$).
- **Ease Factor (EF) Adjustment:** Modifies the difficulty multiplier based on user ratings, preventing "ease hell" for difficult items.

4 Data Layer

The Data Layer is the foundation of the application's state persistence. It is responsible for abstracting the underlying data sources—whether a local database, a network API, or in-memory mock data—from the Domain Layer. This ensures that the business logic remains unaware of how data is stored or retrieved.

4.1 Persistence Strategy

Given the requirement for offline-first functionality, the application utilizes a local SQLite database accessed via the **Room Persistence Library**. Room provides a compile-time verification of SQL queries and an abstraction layer that maps database objects to Kotlin data classes.

4.2 Repository Implementation

The Data Layer implements the `CardRepository` interface defined in the Domain Layer. Two distinct implementations are provided to facilitate different stages of the development lifecycle:

- **FakeCardRepository**: An in-memory implementation used for rapid prototyping and UI testing. It delivers deterministic, hardcoded data (e.g., "Basic Vocabulary" deck) to the ViewModels via the Service Locator pattern. This allows for UI development without requiring a fully populated database.
- **CardRepositoryImpl**: The production-ready implementation that interacts with the Room database DAOs (`CardDao`, `DeckDao`, `ReviewStateDao`). It handles the transformation of `Entity` objects (database rows) into Domain Models.

4.3 Database Schema (Entity Models)

The schema is normalized into three primary tables, represented by the following entity classes:

4.3.1 CardEntity (cards)

Stores the static content of a flashcard.

- **id (PK)**: Unique identifier.
- **deckId (FK)**: Foreign key linking to a specific deck.
- **type**: The card type (VOCAB, CASE, SENTENCE) stored as a String.
- **frontText / backText**: The prompt and answer content.

4.3.2 ReviewStateEntity (review_states)

Stores the dynamic scheduling data for the SRS algorithm. This table is separate from the cards table to decouple content from user progress.

- **cardId (PK/FK)**: One-to-one relationship with `CardEntity`.

- **easeFactor**: A multiplier representing the card's difficulty.
- **intervalDays**: The current spacing interval.
- **dueDate**: Epoch timestamp indicating when the card should next be shown.

4.3.3 DeckEntity (decks)

Stores metadata for grouping cards.

- **id (PK)**: Unique identifier.
- **level**: Difficulty tier (1-3).
- **isPro**: Boolean flag for monetization features.

4.4 Data Access Objects (DAOs)

The application defines interfaces annotated with `@Dao` to generate SQL queries. Key queries include:

- `getDueCards(now)`: `SELECT * FROM review_states WHERE dueDate <= :now`
- `getNewCards(limit)`: `SELECT * FROM cards WHERE id NOT IN (SELECT cardId FROM review_states) LIMIT :limit`

5 Spaced Repetition Logic

5.1 Algorithm Design

The app utilizes a custom scheduler (inspired by SM-2) to calculate the next review interval based on:

- Current Ease Factor
- Repetition Count
- User Rating (Again, Good, Easy)

6 Core Algorithms and Data Flow

This section details the algorithmic logic and data transformations that drive the primary features of the application. The following pseudocode snippets illustrate the interaction between the Domain Use Cases, the Repository layer, and the SRS Scheduler.

6.1 Dashboard Initialization

Upon launching the application, the `HomeViewModel` aggregates data from the repository to present a daily summary. This ensures the user is immediately aware of their pending workload.

```

1 function loadHomeData(currentTime):
2     // 1. Fetch reviews scheduled for today or earlier
3     dueCards = Repository.getDueCards(currentTime)

```

```

4 // 2. Fetch a limited batch of unseen cards
5 newCards = Repository.getNewCards(limit = 10)
6
7 // 3. Retrieve the most recently accessed deck
8 lastDeck = Repository.getAllDecks().firstOrNull()
9
10 // 4. Update the UI State
11 state.reviewsDue = dueCards.count()
12 state.newCardsAvailable = newCards.count()
13 state.lastUsedDeck = lastDeck
14
15
16 return state

```

Listing 1: Load Home Dashboard Data

6.2 Spaced Repetition Scheduler (SRS)

The core retention mechanism is governed by a modified SM-2 algorithm. This function determines the next review date and difficulty interval based on the user's subjective rating of the current card.

```

1 function calculateNextState(currentState, rating, now):
2     // rating is an enum: AGAIN | GOOD | EASY
3
4     if rating is AGAIN:
5         // Reset progress on failure
6         newInterval = 0
7         newEase = max(1.3, currentState.ease - 0.2)
8     else:
9         // Success: Adjust difficulty factor
10        // (Simplified calculation for readability)
11        newEase = calculateEase(currentState.ease, rating)
12
13        // Determine interval based on repetition history
14        if repetitions == 0:
15            newInterval = 1          // 1 day later
16        else if repetitions == 1:
17            newInterval = 6          // 6 days later
18        else:
19            newInterval = currentState.interval * newEase
20
21        // Bonus interval for 'Easy' rating
22        if rating is EASY:
23            newInterval = newInterval * 1.3
24
25        // Calculate future timestamp
26        newDueDate = now + (newInterval * ONE_DAY_MILLIS)
27
28    return ReviewState(newInterval, newEase, newDueDate)

```

Listing 2: Calculate Next Review State

6.3 Study Session Orchestration

The `GetNextCardUseCase` encapsulates the logic for prioritizing content during a study session. It dynamically switches between reviewing due cards and introducing new mate-

rial based on the user's selected study mode.

```
1 function getNextCard(mode, limit):
2     // mode is an enum: REVIEW_ONLY | MIXED | NEW_ONLY
3
4     dueList = Repository.getDueCards(now)
5     newList = Repository.getNewCards(limit)
6
7     if mode is REVIEW_ONLY:
8         return dueList.firstOrNull()
9
10    else if mode is MIXED:
11        // Prioritize reviews, but mix in new cards if available
12        if dueList is not empty:
13            return dueList.first()
14        else:
15            return newList.firstOrNull()
16
17    return null // Session complete
```

Listing 3: Get Next Card for Session

6.4 Review Submission Workflow

When a user submits a rating, the system must persist the new state and immediately prepare the next card. This transactional flow ensures data integrity within the local database.

```
1 function rateCard(cardId, rating):
2     // 1. Retrieve existing history
3     oldState = Database.getReviewState(cardId)
4
5     // 2. Calculate next interval using SRS Logic
6     newState = SrsScheduler.calculate(oldState, rating)
7
8     // 3. Persist updated state to Room Database
9     Database.saveReviewState(newState)
10
11    // 4. Trigger UI update
12    loadNextCard()
```

Listing 4: Rate and Persist Card State

6.5 Deck Retrieval Strategy

Data mapping is essential to convert raw database entities into domain models usable by the UI. This snippet demonstrates the transformation layer within the repository.

```
1 function getDecks():
2     // Query raw entities from SQLite/Room
3     deckEntities = Database.query("SELECT * FROM decks")
4
5     // Map to Domain Models
6     domainDecks = deckEntities.map { entity ->
7         Deck(
8             id = entity.id,
9             name = entity.name,
```

```

10         level = entity.level,
11         isPro = entity.isPro
12     )
13 }
14
15 return domainDecks

```

Listing 5: Fetch and Map Decks

7 User Flow and Interaction

The user experience of the *German Learning App* is designed to be intuitive and focused on daily habit formation. The navigation flow is strictly hierarchical, with the Home Dashboard serving as the central hub for all learning activities.

7.1 User Journey

Upon launching the application, the user is presented with the Home Screen, which provides an immediate summary of their daily goals (reviews due and new cards). From this central point, the user can:

- **Start a Study Session:** By tapping the primary action button, the user enters the Study Screen, where the SRS algorithm determines the sequence of cards.
- **Browse Decks:** Users can explore the library of available content, categorized by difficulty level (e.g., Basic Vocabulary, Case Training).
- **Manage Settings:** Configuration options for audio playback and visual themes are accessible via the settings menu.

7.2 Flowchart Visualization

Figure 2 delineates the complete navigation paths and decision points available to the user. It highlights the cyclic nature of the study session (Review → Rate → Update SRS) which is central to the application’s functionality.

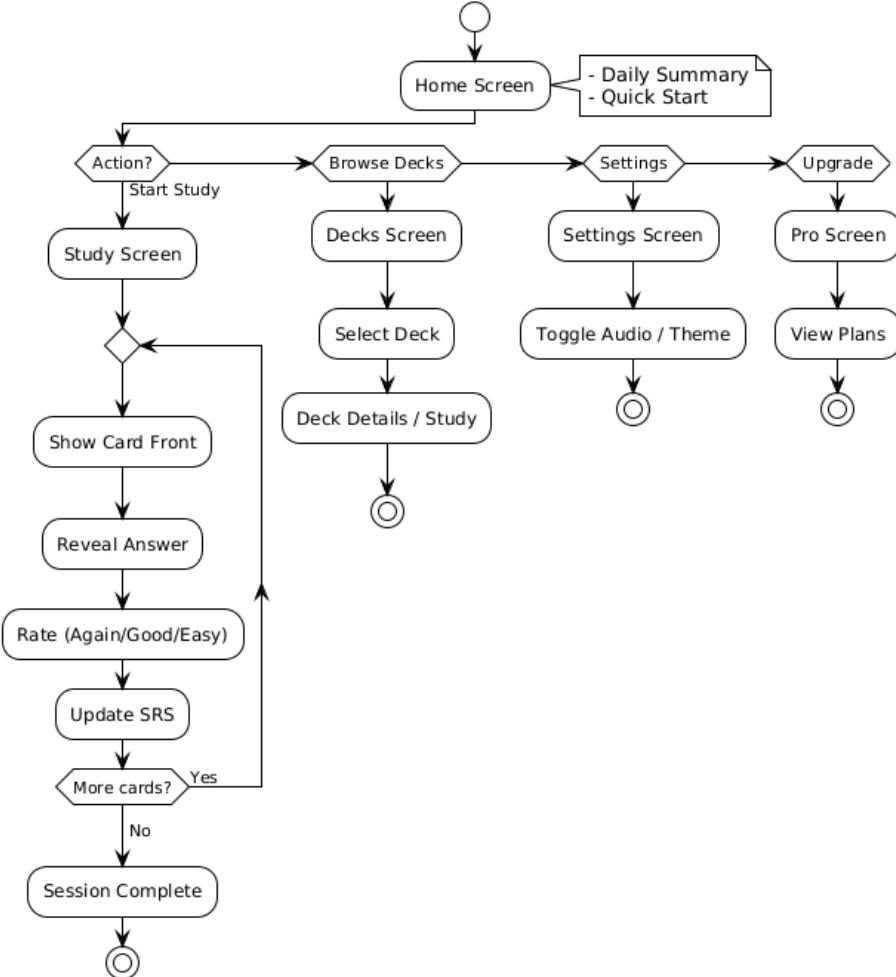


Figure 2: Application User Flow Diagram illustrating navigation paths from the Home Screen.

7.3 Interaction Design

The study interface is designed for minimal friction. Large, clearly labeled buttons for SRS ratings (Again, Good, Easy) allow for rapid input during review sessions. Feedback is immediate; upon rating a card, the next item is presented instantly, maintaining the user’s flow state and maximizing the efficiency of the study time.

8 User Interface (UI)

Built entirely with Jetpack Compose, the UI is state-driven and reactive.

8.1 Navigation Structure

- **Home Screen:** Dashboard with daily summaries.
- **Decks Screen:** Library of available content.
- **Study Screen:** Interactive flashcard interface.
- **Progress & Settings:** User analytics and configuration.

9 Conclusion and Future Work

The current implementation provides a solid foundation for vocabulary acquisition. Future enhancements will include Pro features, cloud synchronization, and AI-generated voice output.