Project documentation

Summary

This project is a modular, strategy-driven flashcard app designed to help learners efficiently memorize vocabulary or concepts. It emphasizes clean separation of concerns, extensibility of presentation behavior, and a simple, focused study experience. The app's architecture allows teams to add new display strategies and learning features with minimal friction.

Core Features

• Flashcard Sessions: Present words or concepts in different modes.
• Display Modes (Strategy Pattern):
  • Term-first view
  • Translation-first view
  • Extensible modes (e.g., phonetic, example sentence, part of speech)
• Clean Mode Switching: Select a display strategy at runtime without altering the underlying data model.
• Lightweight Rendering: Views/controllers request text from a mode and render it.

Architecture

Domain Model

• Word Entity
  • Likely properties: term, translation, and optionally phonetic, examples, partOfSpeech.
  • Designed to be presentation-agnostic. It simply holds data.

Presentation Strategy (Behavioral Pattern)

• FlashcardMode protocol: Defines a single responsibility—how to display a Word.
• Concrete Modes:
  • TermMode: Displays word.term.
  • TranslationMode: Displays word.translation.
  • Potential future: PhoneticMode for word.phonetic, ExampleMode for word.example, etc.
• Benefits:
  • Extensibility: Add new modes without changing existing code.
  • Testability: Validate each mode's behavior independently.

UI Layer

• Views or View Controllers render the display output provided by the active FlashcardMode.
• A central render function or view model composes a Word with a FlashcardMode to produce display text.
• Potential SwiftUI approach:
  • A View binds to a view model that exposes currentWord and currentMode.
  • A toolbar or settings sheet allows switching modes.
  • Animations or transitions provide a pleasant card-flip experience.

State & Navigation

• Session State
  • Tracks current index, total cards, known/unknown status, and progress.
• Mode State
  • Remembers the user's chosen display mode during and across sessions.
• Navigation
  • A top-level screen for deck selection or a simple session launcher.
  • A study screen with the flashcard and controls (reveal, mark correct/incorrect, next).

Design Patterns in This Project

This project uses classic object-oriented design patterns to keep the codebase modular, testable, and easy to extend. From the **FlashcardMode.swift** file, we can already see a clear application of the Strategy pattern for display logic. Below is an overview of patterns that fit the current architecture and ones you can adopt as the project grows.

Strategy Pattern

• Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable at runtime.
• Where it's used: FlashcardMode protocol and its concrete implementations (TermMode, TranslationMode, and potential PhoneticMode).
• Why it's useful here:
  • Decouples the "how to display a Word" from the Word model and the UI.
  • Makes it trivial to add new display behaviors without modifying existing code.
  • Improves testability by isolating each strategy.

Factory Method
• Intent: Encapsulate object creation to avoid scattering init calls and conditionals across the app.
• Where to use: Centralize the creation of FlashcardMode based on user settings or context (e.g., “term-first”, “translation-first”).
• Benefits:
  • Keeps UI and view models free from construction details.
  • Makes it easy to map user preferences to concrete strategies.

Facade Pattern (Unifying flashcard operations)
• Intent: Provide a unified, high-level interface to a set of interfaces in a subsystem. Facade makes the subsystem easier to use.
• Where: An optional “FlashcardFacade” (or “StudyService”) that coordinates word loading, strategy selection, and navigation.
• Why: Simplifies ViewModels and UI by centralizing orchestration. Reduces coupling with internal details (factories, strategies, navigation).

MVVM (Separation of concerns with SwiftUI)
• Intent: Separate UI from business logic and data access.
• Where: SwiftUI Views bind to an ObservableObject ViewModel; the ViewModel uses factories, strategies, and optionally a facade.
• Why: Improves testability, modularity, and clarity. SwiftUI pairs naturally with MVVM.

SOLID Principles in This Project
This document explains how the project aligns with the SOLID principles, why they matter, and how to apply them consistently as the codebase grows. It’s written for direct copy-and-paste into your documentation.
S — Single Responsibility Principle (SRP)
Each module/class should have one reason to change.
• How it applies:
  • FlashcardMode implementations (TermMode, TranslationMode, etc.) each focus exclusively on one display behavior.
  • WordFactory (and its concrete implementations) are solely responsible for creating/loading Word data.
  • ViewModels focus on presentation logic (binding data to the UI), not on data loading or formatting details.
  • Optional FlashcardFacade (if used) centralizes orchestration of words, navigation, and mode selection, keeping Views/ViewModels thin.
• Benefits:
  • Clear responsibilities reduce unintended side effects when making changes.

  • Easier testing: each unit has focused behavior and fewer dependencies.
• Practical tips:
  • If a type starts doing more than one thing (e.g., both loading and formatting words), split it into smaller types.
  • Keep “wiring” or app composition at the edges (e.g., in the app entry point or a small composition layer).

O — Open/Closed Principle (OCP)

Software entities should be open for extension but closed for modification.
• How it applies:
  • Add new display behaviors by creating new FlashcardMode conformers without modifying existing ones.
  • Add new data sources by creating new WordFactory conformers without changing existing factories or consumers.
  • Extend features by composing new objects (e.g., a new FlashcardMode or a decorator) rather than editing core, stable code.
• Benefits:
  • Reduces regression risk since existing, tested code stays unchanged.
  • Encourages plug-in style growth of features.
• Practical tips:
  • Prefer protocol-based abstractions for variation points (display, data, navigation policy).
  • Keep public APIs stable; extend via new types that conform to the same protocols.

L — Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types without breaking correctness.
• How it applies:
  • Any FlashcardMode should work wherever a FlashcardMode is expected; it should return a reasonable string for any valid Word.
  • Any WordFactory implementation should behave consistently: produce valid Word arrays or throw errors in predictable ways.
• Benefits:
  • You can swap strategies and factories freely (e.g., for A/B tests or testing) without surprising failures.
  • Simplifies testing and composition because contracts are reliable.
• Practical tips:
  • Keep protocol contracts explicit (document preconditions/postconditions).
  • Avoid “special” implementations that require hidden assumptions (e.g., a FlashcardMode that only works with specific Word shapes unless clearly documented).

I — Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use.
• How it applies:

  • FlashcardMode is minimal: a single display(word:) method.
  • WordFactory is focused: makeWords() (and maybe small, clearly scoped variants).
  • ViewModels depend on small, purposeful protocols rather than large “god interfaces.”
• Benefits:
  • Smaller protocols are easier to implement, mock, and test.
  • Reduces coupling and accidental dependencies.
• Practical tips:
  • If a protocol starts to grow many unrelated methods, consider splitting it into smaller, cohesive protocols.
  • Keep the method surface area just large enough for the behavior you need right now.

D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions.
• How it applies:
  • ViewModels depend on WordFactory and FlashcardMode protocols, not concrete implementations.
  • An optional FlashcardFacade provides a high-level abstraction that depends on protocols internally, allowing you to swap concrete factories/modes without changing callers.
  • Composition/injection at app boundaries wires concrete types (e.g., LocalJSONWordFactory, TermMode) into abstractions.
• Benefits:
  • Testability: inject mocks/stubs for WordFactory and FlashcardMode.
  • Flexibility: replace data sources or display strategies without refactoring consumers.
• Practical tips:
  • Pass dependencies via initializer injection.
  • Keep concrete type knowledge at the edges (composition root), not inside core logic.

User Guide — How to Run

This guide explains how to set up, build, and run the app locally, plus common tasks like switching flashcard modes and loading data. Copy and paste this into your documentation.

Prerequisites

• Xcode 15 or later (recommended: Xcode 26.2 to match current toolchain)
• macOS compatible with your Xcode version
• iOS Simulator (bundled with Xcode) or a physical iOS device
• Apple Developer Account (only needed for running on a physical device)