What I've L First Module (System Design)

- App Architecture Best Practices
- System Design and Requirements Analysis
- Thinking, Designing, and Drawing Diagrams like a Software Architect
- Dealing with Singletons and Globals: When, Why, How, and Better Alternatives
- SOLID Principles Applied in Practice in Swift
- Clean Architecture
- Modular Design
- Domain-Driven Design (DDD)
- Behavior-Driven Development (BDD)
- Use Case Analysis
- Domain Modeling
- Pair programming
- Effectively Developing Swift Apps Before the Backend/Design Is Ready

---

What I've Learned in the Second Module (Networking)

- Networking Best Practices in Swift
- URLSession, URLSessionDataTasks, URLProtocol, and the URL Loading System in Swift
- Effective Use of 3rd-party Networking Frameworks in Swift (Firebase, Alamofire, etc.)
- Encoding/Decoding JSON API Data to Native Models Using Codable in Swift
- Developing a Testable Networking Layer in Swift
- Four Approaches to Test(-Drive) Network Requests in Swift: End-to-End, Subclass, Protocol-Based Mocking, and URLProtocol Stubbing
- What Many Apps Get Wrong About Reachability and How To Get It Right
- Speeding up Development using frameworks in Xcode
- Speeding up Development by Reducing Debugging Time in Xcode
- Automating a Continuous Integration (CI) Pipeline

- Object-Oriented Programming
- Functional Programming
- Test-Driven Development (TDD)
- Automated Tests
- Unit Testing
- Integration Testing
- End-to-end Testing
- Refactoring
- Version control with git
- Closures and Protocols as Abstractions in Swift
- Identifying, Debugging, and Solving Data Races in Xcode
- Pattern Matching in Swift
- Writing Safe Swift Code by Making Invalid Paths Unrepresentable
- Effective Dependency Management with Dependency Injection in Swift
- Effective Access Control in Swift
- Effective Error Handling in Swift
- Effective Memory Management in Swift
- Automating Memory Leak Detection in Swift
- Configuring Xcode Projects for Running Automated Tests
- Randomizing and Parallelizing Test Execution in Xcode
- Gathering, Understanding, and Improving Code Coverage in Xcode
- Testing Best Practices in Swift
- Common Test Doubles: Spy, Stub, Mock, Fake
- Testing Async Behavior in Swift
- Testing Error Cases in Swift
- Preventing Common Async Bugs in Swift
- Using Swift's Result and Error types

What I've Learned in the Third Module (Persistence)

- Persistence Best Practices in Swift
- Developing a Testable Persistence Layer in Swift
- Choosing Between Persistence Options: In-Memory, UserDefaults, FileSystem,

URLCache, Core Data, and other key persistence frameworks.

- Persisting and Retrieving data with URLCache in Swift
- Persisting and Retrieving data with Codable+FileSystem in Swift
- Persisting and Retrieving data with Core Data in Swift: modeling entities, contexts, concurrency model, and testing techniques
- Concurrency and Threading Best Practices in Swift
- Safely Managing Shared State in Multithreaded Environments
- Designing and Testing Thread-safe Components with DispatchQueue
- Dispath framework in Swift: Serial and Concurrent DispatchQueues, sync, async, barrier flags, and more
- Identifying, Debugging, and Solving Threading Race Conditions
- Thread Safety with Reference and Value Types in Swift
- Composite Reuse Principle (aka Prefer composition over inheritance) applied in Swift
- Protocol-Oriented Programming (POP) in Swift: Protocol inheritance, extensions, composition, and conditional conformance
- Protocol vs. Class inheritance in Swift
- Controlling the Current Date/Time and Other Environment Details During Tests
- Achieving a Healthy Distribution of Unit, Integration, End-To-End, and other testing strategies
- Speeding up Development with App Architecture to Develop Features in Parallel
- Speeding up Development with Iterative Design over Big Upfront Design
- Decoupling Business Logic From Infrastructure Details
- Breaking Down Monoliths into Modules: When, Why and How
- Entities vs. Value objects in Swift
- Data Transfer Objects (DTOs) to Decouple Modules in Swift
- Single Responsibility Principle (SRP) Applied in Swift
- Open-Closed Principle (OCP) Applied in Swift
- Liskov Substitution Principle (LSP) Applied in Swift
- Interface Segregation Principle (ISP) Applied in Swift
- Dependency Inversion Principle (DIP) Applied in Swift

- Don't Repeat Yourself (DRY) Principle Applied in Swift
- Command-Query Separation Principle Applied in Swift
- Functional Core / Imperative Shell Pattern Applied in Swift
- Performing Calendrical Calculations Correctly in Swift
- Domain Specific Languages (DSLs) Applied in Swift
- Achieving High Test Coverage with Triangulation
- Producing a Clean and Stable Codebase History in git
- Designing Side-Effect Free (Deterministic) Core Business Rules
- Maximizing Swift Code Correctness with Single Sources of Truth
- Integration Testing with Real Frameworks (Instead of Mocks)
- Measuring and Improving Test Times with xcodebuild
- Codebase Health Analysis Techniques
- Eliminating Hard-to-Read Nested Code (aka Arrow Code Anti-Pattern)
- Choosing Good Names in Swift

What I've Learned in the Fourth Module (UI + Presentation)

- App UI, UX, Localization, and Presentation Best Practices
- Developing a Clean and Testable UI and Presentation Layers in Swift
- Validating UI Design and Getting Fast Feedback with UI Prototypes
- Working Effectively with Designers
- Efficiently Loading and Presenting Data on Screen
- Efficiently Loading Images in Reusable Cells
- Efficiently Prefetching Images when Cells are Near Visible
- Efficiently Canceling Data Loading Requests to Reduce Data Usage

- MVC: Implementing Best Practices and Variations in Swift
- Identifying and Fixing the Massive View Controller Anti-Pattern
- Multi-MVC Design: Breaking Down Complex Scenes into Tiny Collaborative MVCs
- MVVM: Implementing Best Practices and Variations in Swift
- Identifying and Fixing the Massive View Model Anti-Pattern
- Implementing Stateful and Stateless View Models in Swift
- MVP: Implementing Best Practices and Variations in Swift
- Identifying and Fixing the Massive Presenter Anti-Pattern
- Creating Reusable Cross-platform Presentation Layers with MVVM and MVP in Swift
- Test-driving MVC, MVVM, MVP, and their Variants
- Implementing and Testing Customer-Facing Localized Strings
- Inside-Out vs. Outside-In Development Strategies
- Safely Refactoring Code Backed by Automated Tests and Swift Types (Compiler!)
- Separating Platform-specific and Platform-agnostic Code using Swift Frameworks in Xcode
- Supporting Multiple Platforms on Swift Frameworks in Xcode
- Identifying and Solving Cyclic Dependencies (Retain Cycles) with the Proxy Design Pattern in Swift
- Adapter pattern Applied in Swift: Enabling components with incompatible interfaces to work together seamlessly
- Decorator Pattern Applied in Swift: Extending behavior of individual objects without changing their implementation
- Safely Adding Tests to Legacy Code
- Creating Reusable Components with Generics in Swift

What I've Learned in the Fifth Module (Main Composition)

- App Composition Best Practices in Swift
- Developing a Clean and Testable Composition Layer in Swift
- Composing Swift Modules to Form a Fully Functional App in Xcode
- UI Testing Best Practices in Swift
- Writing Reliable Automated UI Tests by Controlling Network and App State with Launch Arguments and Conditional Compilation Directives
- Acceptance Testing Best Practices in Swift
- Replacing UI Acceptance Tests with Significantly Faster Integration Acceptance Tests.
- Snapshot Testing Best Practices in Swift: Automatically Validating the App's UI
- Speeding up Snapshot Tests by Rendering Views Without Running The App
- Supporting Dark Mode
- Strategy Pattern Applied in Swift
- Composite Pattern Applied in Swift
- Composition Root Design Pattern Applied in Swift
- Eliminating Memory Leaks and Threading Issues by Managing Memory and Threading in the Composition Layer
- Simulating App Launch and State Transitions During Tests
- Testing Methods You Cannot Invoke
- Testing System Classes with Private Initializers
- Organizing Modular Codebases with Horizontal and Vertical Slicing
- Organizing the Codebase into Independent Frameworks, Packages or Projects
- Continuous Integration Best Practices
- Continuous Delivery Best Practices
- Continuous Deployment Best Practices
- App Store Deployment Best Practices
- Deploying Builds to App Store Connect Manually
- Deploying Builds to App Store Connect Automatically with a Continuous Delivery & Deployment (CD) pipeline
- Effectively using the Combine Framework in Swift: Implementation and Tests
- Eliminating Duplication from Design Patterns with Universal Abstractions using the Combine Framework

What I've Learned in the Sixth Module (Navigation + Advanced Patterns)

- Best Practices for Navigation in Swift
- Developing a Clean and Testable Navigation Layer in Swift
- Navigating Between Independent Features Without Breaking Modularity or Introducing Common Anti-Patterns
- App Composition and Dependency Injection Best Practices
- The Object-oriented vs. the Functional way of Composing Components in Swift
- Dependency Injection and Dependency Rejection Patterns in Swift
- Decoupling Feature Modules without Introducing Duplication
- Refactoring Code Backed by Tests and the Compiler
- Null Object Pattern in Swift
- Creating Reusable Generic Components in Swift
- Best Practices for Creating Reusable Presentation Logic
- Localized Date Formatting Best Practices
- Writing Fast and Reliable Automated Tests by Controlling the Current Date, Locale, Calendar, and other Environment Details in Swift
- Best Practices for Creating Reusable UI Components
- Creating a Shared UI Module without Breaking Modularity
- Creating UI Elements Programmatically
- Solving Memory Leaks with the Memory Graph Debugger in Xcode
- Migrating to Diffable Data Sources
- Scaling Fonts with Dynamic Type
- Using Snapshot Tests Effectively
- Best Practices for Modular Feature Composition
- Developing and Testing the Composition Root
- Using Autoreleasepools in Swift
- Releasing Autoreleased Instances During Tests
- Choosing Between Diffrent Dependency Lifestyles: Singleton, Transient, and Scoped

- Common API Pagination Methods and Best Practices in Swift
- Keyset Pagination with Caching Strategy in Swift
- Pagination UI/UX Best Practices
- Unit, Snapshot, Integration, and Acceptance Testing Best Practices in Swift
- Logging Best Practices in Swift
- Logging, Profiling, and Optimizing Infrastructure Services
- Monitoring Debug and Release Builds in a Clean Way
- Async Injection: Decoupling the Domain from Infra Async Details
- Eliminating Async Boilerplate & Nested Callbacks
- Implementing Type Erasure in Swift
- Composing Async Operations with Combine
- Composing Modules with Built-in Combine Operators
- Managing Threading in a Clean Way with Combine Schedulers
