Real-World Application of SOLID, DRY, and KISS in iOS Development

Applying SOLID, DRY, and KISS Principles in iOS Development with Swift

---

Introduction

As the complexity of iOS applications rises alongside user expectations, developing readable, maintainable, and scalable code becomes a non-negotiable requirement. This imperative leads iOS developers toward adopting established software engineering principles such as **SOLID**, **DRY** (Don't Repeat Yourself), and **KISS** (Keep It Simple, Stupid). These principles are foundational in crafting robust Swift code that helps prevent common pitfalls like tightly coupled modules, code duplication, and unnecessary complexity.
This report presents an in-depth examination of how these principles specifically apply to iOS development using Swift, supported by practical examples and real-world scenarios. We will explore their impact on login flows, reusable UI components, service layers, and persistence solutions, and provide guidance on integrating these principles into everyday iOS engineering practice. Each principle will be defined, dissected, and demonstrated with Swift-based code patterns, accompanied by discussions of their practical benefits and common challenges encountered in real projects.

Summary Table: Principles and Their Benefits in iOS (Swift) Development

Principle	Core Idea	Practical Benefits in iOS Development
SRP (Single Responsibility)	A class/module has one reason to change	Easier maintenance, testability; fewer merge conflicts
OCP (Open/Closed)	Open for extension, closed for modification	Easier to add features, fewer bugs, maintain stable tested code
LSP (Liskov Substitution)	Subclasses/implementations usable as base classes	Enforces correct behavior, safe abstraction, simpler polymorphism
ISP (Interface Segregation)	No forced implementation of unused interfaces	Finer-grained protocols, improved flexibility, lower coupling
DIP (Dependency Inversion)	High-level modules depend on abstractions, not details	Improved modularity, testability, easier swapping of implementations

DRY (Don’t Repeat Yourself)	Eliminate code duplication	Less maintenance, reduced bugs, code reuse through functions/extensions
KISS (Keep It Simple, Stupid)	Simplicity over complexity	Better code quality, faster development, easier onboarding

Each row in the table summarizes how the respective principle directly addresses common pain points in Swift app engineering, setting the stage for the principle-specific sections below[2][4][6][7].

The SOLID Principles in Swift for iOS Development

Overview

The SOLID principles, introduced by Robert C. Martin ("Uncle Bob"), are five object-oriented design heuristics renowned for enhancing code modularity, clarity, and adaptability. Swift, as a multi-paradigm language with powerful protocol and value-semantics features, makes adopting these principles particularly effective for iOS.
Let's break down each principle with real-world Swift code and discuss their relevance for iOS developers.

Single Responsibility Principle (SRP)

Definition:
A class or module should have **one**, and only one, reason to change. This means each class should do one thing only, encapsulating just a single piece of functionality.
Why It Matters in iOS:
iOS View Controllers are notoriously prone to violating SRP, becoming "Massive View Controllers" when overloaded with UI logic, network calls, data parsing, and more. Adhering to SRP keeps code manageable and testable.
Example: Refactoring an iOS Login Flow
Anti-Pattern (SRP Violation):

```
final class LoginViewController: UIViewController {
@IBOutlet var emailField: UITextField!
@IBOutlet var passwordField: UITextField!

// ... setup code ...

@IBAction func signInTapped() {
// Networking, validation, and UI response all in one place
let email = emailField.text ?? ""
let password = passwordField.text ?? ""
signIn(email: email, password: password)
```

```
}

private func signIn(email: String, password: String) {
// Network request
// Parse response
// Show alerts
// Log analytics
}
}
```

Refactored (SRP Compliant):

```
// Handles the network call
struct AuthService {
func login(email: String, password: String, completion: @escaping (Result<User, Error>) -> Void) {
// Networking code
}
}

// Only for decoding
struct UserDecoder {
func decode(_ data: Data) -> User? {/* ... */}
}

// Handles UI only
final class LoginViewController: UIViewController {
var authService: AuthService!

@IBAction func signInTapped() {
guard let email = emailField.text, let password = passwordField.text else { return }
authService.login(email: email, password: password) { [weak self] result in
DispatchQueue.main.async {
switch result {
case .success(let user):
self?.showWelcome(for: user)
case .failure(let error):
self?.showError(error)
}
}
}
}
}
```

Discussion:

Here, each class or struct addresses a single concern: the ViewController responds to the user,

AuthService handles authentication calls, and UserDecoder performs decoding. This separation makes each module easier to test, change, or reuse without unintended side effects[9][11].

SRP in UI Components:

For reusable UI (like a custom AlertView), SRP ensures that the component only handles presentation logic. Data sources, theming, and business logic are handled elsewhere.

Benefits:

- Easier unit testing
- Improved readability
- Lower regression risk on change
- Fewer merge conflicts in teams

Open/Closed Principle (OCP)

Definition:

Software entities (classes, modules, functions) should be **open for extension, but closed for modification**. You should be able to add new behaviors without altering existing tested code.

Protocol-Based OCP in Swift:

Before (violates OCP):

Suppose you have hardcoded notification logic:

```
class NotificationManager {
func send(type: String, message: String) {
if type == "email" { /* ... */ }
else if type == "sms" { /* ... */ }
}
}
```

Adding a new notification type requires modifying the original class-violating OCP.

After (OCP-compliant):

```
protocol Notifiable {
func send(message: String)
}

class EmailNotification: Notifiable {
func send(message: String) { /* ... */ }
}
class SMSNotification: Notifiable {
func send(message: String) { /* ... */ }
}

class NotificationManager {
func sendNotification(_ notification: Notifiable, message: String) {
```

```
notification.send(message: message)
}
}

// Usage:
let manager = NotificationManager()
manager.sendNotification(EmailNotification(), message: "Welcome!")
```

Discussion:
Now, to add a new notification type (e.g., PushNotification), simply create a new class conforming to Notifiable. The existing, tested code in NotificationManager doesn't change[13][1].
OCP in Persistence Layer:
With protocols and dependency injection, you can abstract persistence:

```
protocol UserPersistence {
func save(user: User)
func loadUsers() -> [User]
}

class CoreDataUserPersistence: UserPersistence { /* ... */ }
class FileUserPersistence: UserPersistence { /* ... */ }
```

You can hand different persistence strategies to your controllers without modifying their code.
Benefits:

- Fewer bugs during feature updates/additions
- Potential for plugins/extensions (think payment gateways, themes, etc.)
- Increased stability
- Simpler A/B testing

Liskov Substitution Principle (LSP)

Definition:
Objects of a superclass should be replaceable with objects of a subclass **without altering the correctness** of the program. Or, in Swift, any type conforming to a protocol should fulfill its contract so it can be safely used in place of the base type.
Classic LSP Scenario in Swift:
Suppose you have:

```
class Bird {
func fly() { print("Flying") }
}
class Ostrich: Bird {
override func fly() {
fatalError("Ostriches can't fly!")
```

```
}
}
```

Violation:
Passing an Ostrich to code expecting a Bird will crash if fly() is called[11].
How to Fix? Use Protocols:

```
protocol Flyable {
func fly()
}
class Sparrow: Flyable { func fly() { print("Flying!") } }
// Ostrich does NOT conform to Flyable

func sendFlying(_ bird: Flyable) { bird.fly() }
```

Discussion:
Instead of "watering down" your protocol so all potential conformers fit (which leads to empty or crashing implementations), define more granular contracts and only conform where appropriate.
LSP in Networking/Error Handling:
Subclasses of Error or custom error types must fulfill the required properties and methods so that error-handling routines can uniformly handle all error cases without switching logic based on concrete types[15][16].
Benefits:

- Promotes safer polymorphism
- Avoids runtime exceptions due to bad substitutions
- Cleaner code without special-case logic or typecasting

Interface Segregation Principle (ISP)

Definition:
Clients should **not be forced to depend upon interfaces they do not use.** In Swift, this maps to using specific, small protocols rather than general, "fat" ones.
Fat Protocol Example:

```
protocol Worker {
func work()
func eat()
}
class Developer: Worker { func work() { /*...*/ }; func eat() { /*...*/ } }
class Robot: Worker { func work() { /*...*/ }; func eat() { /* robots don't eat */ } }
```

Here, Robot is forced to implement eat(), which is nonsensical.
Segregated Protocols:

```
protocol Workable { func work() }
protocol Eatable { func eat() }

class Developer: Workable, Eatable { /* ... */ }
class Robot: Workable { /* ... */ }
```

Now types only conform to the protocols relevant to them.

ISP in iOS-Gesture Handlers:

Suppose:

```
protocol GestureProtocol {
func didTap()
func didDoubleTap()
func didLongPress()
}
```

Not all buttons handle all gestures. Instead:

```
protocol SingleTap { func didTap() }
protocol DoubleTap { func didDoubleTap() }
protocol LongPress { func didLongPress() }

class SuperButton: SingleTap, DoubleTap, LongPress { /* ... */ }
class DoubleTapButton: DoubleTap { /* ... */ }
```

Discussion:

With protocol composition, Swift lets you have multiple, focused interfaces, and your types only implement what makes sense[18][1][12].

Benefits:

- Lower coupling
- Finer-grained testability
- Simpler code for implementers

Dependency Inversion Principle (DIP)

Definition:

High-level modules should not depend on low-level modules; both should depend on abstractions. In practice, this translates into using protocols and dependency injection.

Common iOS DIP Violation:

```
// Bad: High level depends on concrete
class UserManager {
var database = Database()
func saveUser() {
database.save(user: ...)
```

```
}
}
```

Any change to Database ripples through UserManager.

DIP-Compliant with Protocol Injection:

```
protocol UserStorage {
func save(user: User)
}

class Database: UserStorage { /* ... */ }
class InMemoryStorage: UserStorage { /* ... */ }

class UserManager {
private let storage: UserStorage
init(storage: UserStorage) {
self.storage = storage
}
func saveUser(_ user: User) {
storage.save(user: user)
}
}
```

Now, UserManager's behavior can change just by passing in a new UserStorage implementation (file, database, remote API, mock for testing, etc.).

DIP in Service Layers:

For networking or API services, always depend on a protocol rather than a concrete class, making it trivial to swap the implementation or to provide a mock for tests[20][22][23].

Using Dependency Injection Frameworks:

In advanced scenarios or large apps, consider frameworks like **Swinject** or lightweight property-wrapper-based solutions for scalable injection. See Injector for a Swift-focused, SPM-compatible solution, which supports singleton and transient lifecycles as well as named registrations[24].

Benefits:

- Decouples modules, enhancing flexibility and reuse
- Dramatically improves unit testability
- Facilitates mocking/stubbing in tests
- Speeds refactoring and onboarding

DRY Principle in Swift: Don’t Repeat Yourself

Core Idea

The **DRY** principle-"Don't Repeat Yourself"-instructs developers to avoid duplicating logic or code. Every distinct piece of logic should appear exactly once in the codebase. This reduces bugs, simplifies updates, and improves maintainability[4][5].

Implementation in iOS/Swift

1. Functions for Shared Logic

If you use similar code in multiple places (e.g., formatting, validation, alert presentation), fold it into a function:

func showAlert(on vc: UIViewController, title: String, message: String) { /* ... */ }

Whenever you need to display an alert, you call this function instead of duplicating the alert-creation code everywhere.

2. Extensions for Utility

Rather than copy-pasting date formatting code, use an extension:

```
extension Date {
func formattedMedium() -> String {
let formatter = DateFormatter()
formatter.dateStyle = .medium
return formatter.string(from: self)
}
}
```

Now, any date in your codebase can use .formattedMedium().

3. Computed Properties for Reusable Checks

Before (non-DRY):

```
if user.age > 17 { ... }
if user.age > 17 { ... }
```

After:

```
struct User {
let age: Int
var isAdult: Bool { age > 17 }
}
```

Now, use if user.isAdult { ... } everywhere.

4. Generics for Type-Agnostic Logic

Before:

```
func printIntArray(_ array: [Int]) { array.forEach { print($0) } }
func printStringArray(_ array: [String]) { array.forEach { print($0) } }
```

After:

```
func printArray<T>(_ array: [T]) { array.forEach { print($0) } }
```

This single function works for arrays of any type.

5. Protocol Extensions

If several types (e.g., multiple view models) need a logging function:

Before:

```
class UserViewModel { func log() { print("UserViewModel log") } }
class ProductViewModel { func log() { print("ProductViewModel log") } }
```

After:

```
protocol Loggable { func log() }
extension Loggable {
func log() { print("\(Self.self) log") }
}
class UserViewModel: Loggable {}
class ProductViewModel: Loggable {}
```

All conforming types receive a default implementation[3].

DRY in iOS UI

Reusable UI Components:
Define reusable views as custom UIView, UIStackView, or SwiftUI View structs. For example, encapsulate an address form as its own view, and use it wherever address input is required[26][27].
Reusable Table View Cells:
Create subclasses or structures with reusable identifiers.

When NOT to DRY

Not all repetition is bad-if two similar but *meaningfully different* functions may diverge in the future, sometimes it's clearer to repeat a line or two rather than overgeneralizing prematurely[3].

Benefits

- Fixes and enhancements are performed in one location, reducing error risk
- Increases codebase clarity and conciseness
- Eases internationalization/localization
- Reduces copy-paste errors

KISS Principle: Keep It Simple, Stupid

Core Idea

The **KISS** principle advocates for straightforward, clear, and uncomplicated solutions. Code should be as simple as possible while still accomplishing the objective-avoiding over-engineering or needless abstraction[28][5].

Implementation in iOS/Swift

1. Prefer Simpler Logic

Choose code that is easily graspable at a glance. For instance:

label.textColor = isSuccess ? .blue : .red

...is simpler and easier to scan than a verbose if/else block for coloring labels.

2. Avoid Over-Abstracting

Do not create elaborate inheritance trees when protocol composition or simple structs suffice. For example, don't build a full-blown data persistence service layer if UserDefaults or a small persistence wrapper is all you need.

3. Use Standard Library Features

Swift's standard library and Foundation already provide efficient, tested constructs-use them! Avoid reinventing wheel via home-brewed caches, collections, or parsing unless necessary.

4. Write Idiomatic Swift

Use computed properties, functional tools (map, filter, reduce), and trailing closures as appropriate, without extra ceremony.

5. Favor Value Types When Possible

Structs and enums offer performance and clarity benefits for many domain models; avoid classes where mutability or inheritance isn't required.

6. Incremental Refactoring

Refactor only when needed, not in anticipation of every possible future requirement. "You Aren't Gonna Need It" (YAGNI) often accompanies KISS.

7. Use Expressive Naming

Choose short but descriptive names for variables, methods, and classes/structs. Avoid vague names or misleading abbreviations.

KISS in Everyday iOS Development

- Write concise view controllers; move out logic into helpers or models.

- Use concise, expressive error handling.
- Avoid giant configurations-encapsulate defaults, use property wrappers if needed.
- When integrating a third-party service, start with their minimal working example, then customize as *truly required*.

Benefits

- Faster code comprehension and onboarding for new developers
- Reduced bug surface area
- Simpler, less expensive testing and maintenance
- More robust code under change; fewer cross-module breakages

---

Concrete Applications: Principles in Action

1. Applying SOLID and DRY in a Login Flow

Scenario:
Building a login screen requiring user authentication, feedback, and navigation.
SRP:

- LoginViewModel handles only form validation and requests, not UI.
- AuthService manages network interaction (SRP promotes separation).

OCP:

- AuthService uses protocols: can swap authentication mechanism (e.g., email, external OAuth) via different conformers (OCP).

DIP and DRY:

- LoginViewController receives its dependencies via initializer injection or property injection-enables easy unit testing.
- Observers for login state use protocols, letting you inject mocks for tests.

KISS:

- Avoid “clever” architectures until scaling actually reveals complexity.
- Only build as much abstraction as current requirements justify.

Swift Example:

```
protocol AuthenticationService {
func login(email: String, password: String, completion: @escaping (Result<User, Error>) -> Void)
}

class APIAuthService: AuthenticationService {
func login(email: String, password: String, completion: @escaping (Result<User, Error>) -> Void) {
```

```
// Make network call, handle success/failure, call completion
}
}

class LoginViewModel {
private let authService: AuthenticationService
init(authService: AuthenticationService) {
self.authService = authService
}

func login(email: String, password: String, completion: @escaping (Bool) -> Void) {
authService.login(email: email, password: password) { result in
completion(result.isSuccess)
}
}
}

// Inject mock or real service easily for testability.
```

(You can test, swap backends, or update the networking layer with minimal changes.)

---

2. Reusable UI Component Library in Swift/SwiftUI

- **DRY:** Encapsulate custom UI (e.g. Button, Card, TextField) as reusable View structs or UIView subclasses.
- **SRP:** Each custom view's responsibility is limited to display-no data fetching or unrelated side effects.
- **OCP:** Components are extensible via protocols or composition-can add theming, animation, etc., without changing existing code.
- **KISS:** Default to simple, parameter-driven custom views; avoid premature, large-scale theming engines unless needed.

SwiftUI Example:

```
struct CustomButton: View {
let title: String
let action: () -> Void
var backgroundColor: Color = .blue

var body: some View {
Button(action: action) {
Text(title)
.foregroundColor(.white)
.padding()
```

```
.background(backgroundColor)
.cornerRadius(10)
}
}
}
```

Use CustomButton(title: "Sign In", action: {...}) in multiple places-no copy-paste of UI logic or styling.

---

3. Service Layer and Networking

- Use **DIP**: Service layer is defined as a protocol (e.g., APIService); actual implementation can be REST/GraphQL, mocked, etc.
- **OCP**: Add a new endpoint/feature by adding new implementers, not by modifying existing ones.
- **SRP/DRY**: Move shared parsing, authentication, and networking logic into helpers/extensible objects.

Example:

```
protocol UserService {
func fetchUser(id: String, completion: @escaping (Result<User, Error>) -> Void)
}

class NetworkUserService: UserService {
func fetchUser(id: String, completion: @escaping (Result<User, Error>) -> Void) {
// Networking code...
}
}
```

View models/controllers **depend on UserService**, not on NetworkUserService itself (enabling easy test/mocking-see unit testing below)[31][21][19].

---

4. Data Persistence Layer with SOLID and Protocol-Oriented Design

- Use protocols for persistence (PersistenceStore), so that storage tech can be swapped (UserDefaults, Core Data, new SwiftData, JSON file, cloud, etc.).
- **LSP:** All persistence implementations must adhere to the contract (methods, behavior).
- **OCP:** Add new storage types via new conformers.
- **DIP:** High-level code relies on the protocol.
- **KISS/DRY:** Only persist the minimum data required in as simple a fashion as possible for the domain.

Swift Example:

```
protocol TodoStorage {
func save(todos: [Todo])
func loadTodos() -> [Todo]
}

class UserDefaultsTodoStorage: TodoStorage { /* ... */ }
// Add CoreDataTodoStorage, MockTodoStorage as needed.
```

Swap storage engines-no code change required to consumer code.

---

5. Dependency Injection Frameworks

In projects with complex dependency graphs or modules, using Swinject or a custom SPM dependency injector can help manage injections without excessive boilerplate. Leverage property wrappers or containers for auto-injection and scope control (singleton/transient).
Example:

```
import Swinject

let container = Container()
container.register(AuthService.self) { _ in APIAuthService() }

let authService = container.resolve(AuthService.self)!
```

KISS Note: Only introduce DI containers when the complexity justifies it-prefer manual injection for smaller apps.

---

6. Protocol-Oriented Programming (POP) & SOLID

Swift's POP paradigm amplifies SOLID compliance:

- Prefer protocols over base classes for contracts.
- Use protocol extensions for default logic (DRY).
- Define small, focused protocols (ISP).
- Enable dependency injection via protocols (DIP).

---

7. Testing, Maintainability, and Scalability

- **Unit Testing**:
  SRP/DIP make unit testing easy-inject mocks for dependencies, test in isolation. Use XCTest (built-in to Xcode/Swift) for test discovery, setup, and teardown[34][35].
- **Mocking & Stubbing**:

Depend on protocols, not concrete classes-swap real implementations for test doubles in tests.

- **Maintainability**:
  KISS and SOLID keep the codebase small and flexible; DRY ensures you update logic in one place only.
- **Scalability**:
  OCP and DIP allow you to add new modules, microservices, or UI paradigms (UIKit, SwiftUI) with less risk of regressions or rewrites.

Best Practices & Pitfalls

Best Practices:

- Begin with the simplest, clearest solution possible (KISS).
- Refactor and extract protocols as needs arise-not before.
- Always abstract persistence/networking behind protocols to future-proof your app.
- Use dependency injection for all service-like objects-manual or via a container.
- Avoid "fat" protocols; keep interfaces and protocols focused on one job.
- Prefer value types (structs, enums) with protocol conformance for domain models.
- Write unit tests for every module using dependency injection and mocks.

Common Mistakes to Avoid:

- Over-abstraction or overuse of patterns (e.g., unnecessary factories or DI containers in small apps).
- Making protocols too broad, causing unwanted method implementations (ISP violation).
- Introducing complexity in anticipation of requirements that may never arrive (YAGNI-You Aren't Gonna Need It).
- Failing to update protocol contracts or default implementations when requirements change.
- Omitting dependency injection leading to tightly coupled and untestable modules.
- Copy-pasting code between features/contexts instead of extracting reusables (violates DRY).

Conclusion

The SOLID, DRY, and KISS principles-when adapted thoughtfully to the Swift and iOS context-yield Swift codebases that are modular, readable, maintainable, and scalable. By focusing on single responsibilities, extensibility through protocols, decoupling via dependency inversion, and systematic code reuse, you sidestep a host of pitfalls, from massive view controllers to rigid, fragile service layers.

Simplicity (KISS) should underscore every architectural or refactoring decision. Start simple-advance to abstract and generalize only as genuine requirements develop. When combined with robust protocols, practical dependency injection, and reusable code patterns, these principles form the architecture underlying the best modern iOS applications.
By consistently applying SOLID, DRY, and KISS in your Swift projects, you can confidently deliver iOS experiences that are robust, maintainable, and elegantly simple, ready to grow and evolve along with user needs and technological change.

References:
Practical Swift code examples and explanations have been synthesized and adapted from a wide range of authoritative sources, including developer blogs, open-source Swift repositories, community tutorials, and Apple's own documentation[36][8][11][13][16][17][21][38][4][2][29][40][25][27][30][31][23][24][43][33][35]. The referenced code demonstrates the tangible benefits of these principles throughout every layer of a modern iOS app, from user-facing UI to hidden persistence and service infrastructure.

References (43)

1. *KISS Principle in Software Development - GeeksforGeeks*. https://www.geeksforgeeks.org/software-engineering/kiss-principle-in-software-development/
2. *SOLID Principles in Swift: A Practical Guide - DEV Community*. https://dev.to/marwan8/solid-principles-in-swift-a-practical-guide-1fgk
3. *DRY Principle in Software Development - GeeksforGeeks*. https://www.geeksforgeeks.org/software-engineering/dont-repeat-yourselfdry-in-software-development/
4. *Protocol Oriented Programming in Swift - Swift Anytime*. https://www.swiftanytime.com/blog/protocol-oriented-programming-in-swift
5. *Introduction to Protocol-Oriented Programming (POP) in Swift*. https://codezup.com/swift-protocol-oriented-programming-introduction/
6. *SOLID explained with iOS examples - DEV Community*. https://dev.to/ishouldhaveknown/solid-explained-with-ios-examples-28ni
7. *SOLID Principles in Swift: Open/Closed Principle - DEV Community*. https://dev.to/bionik6/solid-principles-in-swift-open-close-principle-3f3b
8. *The SOLID Principles with Practical Examples in Swift*. https://stackademic.com/blog/the-solid-principles-with-practical-examples-in-ios-swift
9. *#2 SOLID - Open Close Principle* . https://dev.to/bibinjaimon/2-solid-open-close-principle-swift-ios-development-6ek
10. *Liskov Substitution Principle in Swift* . https://www.youtube.com/watch?v=a4qVpupywzo
11. *#3 SOLID - Liskov Substitution Principle* . https://dev.to/bibinjaimon/3-solid-liskov-substitution-principle-swift-ios-development-3042

12. *Interface Segregation Principle in Swift* . https://freedium.cfd/1778bab4452b
13. *Understanding Dependency Injection in Swift* . https://codezup.com/understanding-dependency-injection-swift/
14. *Swift Dependency Inversion through Protocol - mobidevtalk.com*. https://mobidevtalk.com/swift-dependency-inversion-through-protocol/
15. *Complete Guide to Dependency Injection in Swift*. https://www.swiftanytime.com/blog/dependency-injection-in-swift
16. *GitHub - lucasdeprit/Injector: A lightweight and easy-to-use dependency ...*. https://github.com/lucasdeprit/Injector
17. *Key Principles In Software - GeeksProgramming*. https://geeksprogramming.com/key-principles-in-software-and-acronyms/
18. *Clean, Reusable Swift Code Using DRY Principle - Swift Shorts*. https://swiftshorts.com/clean-reusable-swift-code-using-dry-principle/
19. *Creating A Swiftui Component Library For Reusable Ui Elements*. https://peerdh.com/blogs/programming-insights/creating-a-swiftui-component-library-for-reusable-ui-elements
20. *Creating a Custom Swift UI Component Library - codezup.com*. https://codezup.com/swift-ui-component-library/
21. *The KISS Principle - Codefinity*. https://codefinity.com/blog/The-KISS-Principle
22. *SwiftUI Data Persistence in 2025: SwiftData, Core Data, AppStorage ...*. https://dev.to/swift_pal/swiftui-data-persistence-in-2025-swiftdata-core-data-appstorage-scenestorage-explained-with-5g2c
23. *Dependency Inversion Principle in Swift - DEV Community*. https://dev.to/kevinmaarek/dependency-inversion-principle-in-swift-1h15
24. *Dependency Inversion - A Little Swifty Architecture - Clean Swift*. https://clean-swift.com/dependency-inversion-a-little-swifty-architecture/
25. *Discover Unit Testing on Swift by Sundell*. https://www.swiftbysundell.com/discover/unit-testing/
26. *Swift Testing: Comprehensive Unit Tests for iOS Apps*. https://codezup.com/swift-testing-unit-tests-5/
27. *SOLID Principles for iOS Apps - Kodeco*. https://www.kodeco.com/21503974-solid-principles-for-ios-apps
28. *SOLID Principles in Swift: Single Responsibility Principle*. https://dev.to/bionik6/solid-principles-in-swift-single-responsibility-principle-4jcd
29. *#4 SOLID - Interface Segregation Principle - DEV Community*. https://dev.to/bibinjaimon/4-solid-interface-segregation-principle-swift-ios-development-4e2e
30. *Dependency injection and IoC in iOS and Swift* . https://danielsaidi.com/blog/2014/09/04/ioc-in-ios
31. *KISS principle - Wikipedia*. https://en.wikipedia.org/wiki/KISS_principle
32. *Secure iOS Login & Registration with Swift & UIKit Guide*. https://mojoauth.com/blog/ios-secure-login-registration-swift-guide/

33. *Building a Custom UI Component Library with Swift and UIKit*. https://codezup.com/custom-ui-component-library-swift-uikit/
34. *Creating a Service Layer in Swift - Medium*. https://livefront.com/writing/creating-a-service-layer-in-swift/
35. *Protocol-oriented Programming in Swift* . https://www.toptal.com/swift/introduction-protocol-oriented-programming-swift
36. *Getting started with Unit Tests in Swift - SwiftLee*. https://www.avanderlee.com/swift/unit-tests-best-practices/