SWIFTUI PREP FOR WISE PAIR PROGRAMMING

Only what you need when iOS interviewers ask follow-ups

Connected to the 5 problems you're practicing

HOW SWIFTUI WILL COME UP IN THIS INTERVIEW

Based on the email: "*pair programming with two iOS developers... collaboration and architecture designing skills*"

SwiftUI won't be the main coding task. But it WILL come up in 3 ways:

1. **First 5 min discussion:** "What's your experience with SwiftUI?"
2. **During coding:** They'll notice if you naturally think in reactive patterns
3. **After coding:** "How would you integrate this into our app?"

This file covers exactly those 3 moments — nothing more.

PART 1: THE DISCUSSION QUESTIONS (First 5-10 min)

Q: "What's your experience with SwiftUI?"

> "I've been driving SwiftUI adoption on my team at PayPal. We're migrating from UIKit incrementally — new features in SwiftUI, legacy screens wrapped with UIHostingController. I'm comfortable with the full SwiftUI stack: declarative views, property wrappers for state management, Combine for reactive data flow, and async/await integration with .task modifier.
> I know Wise moved fully to SwiftUI with Combine back in 2022, so I'd be working in my preferred framework from day one."

Q: "SwiftUI or UIKit? What's your opinion?"

> "SwiftUI for everything new. It's declarative, less boilerplate, built-in accessibility and dark mode support, and Apple's entire investment is there. The main limitation was maturity — but iOS 16+ SwiftUI handles almost everything.
> Where I still respect UIKit is complex custom interactions — advanced collection view layouts, certain gesture handling, and interop with older SDKs. But those are edge cases. For a fintech app's core flows — transfers, balances, cards — SwiftUI is the right choice."

Q: "How do you handle architecture with SwiftUI?"

> "MVVM is the natural fit. SwiftUI views are the V — lightweight structs that declare what the UI should

> look like. The ViewModel is an ObservableObject with @Published properties — it transforms domain data for display and handles user actions. The Model layer is pure Swift types and business logic, completely independent of any UI framework.
> The key principle: the ViewModel never imports SwiftUI. It's plain Swift, fully testable with XCTest. The View observes the ViewModel through @StateObject or @ObservedObject, and SwiftUI handles the re-rendering automatically when @Published properties change."

Q: "How do you manage navigation in SwiftUI?"

> "For simple flows, NavigationStack with NavigationLink and .navigationDestination works well. For complex multi-step flows like a transfer — amount → recipient → payment method → review → confirmation — I'd use a coordinator pattern or a state-driven approach where the ViewModel controls which step is active.
> At Wise with Dynamic Forms, navigation is likely server-driven anyway — the backend tells the app which screen comes next, and the mobile client renders it."

PART 2: CONNECTING YOUR CODING SOLUTIONS TO SWIFTUI

For each of the 5 problems, here's how to answer "how would this work in the app?"

Circuit Breaker → SwiftUI Integration

After you code the CircuitBreaker, they might ask: "*How would the app use this?*"

```swift
```

```
// The CircuitBreaker you coded sits INSIDE a networking layer.
// The ViewModel uses it through a service — it never knows about circuit breakers directly.

// Service layer (wraps your CircuitBreaker)
class TransferService: ObservableObject {
    private let client: WebClient  // Your WebClient with CircuitBreaker inside

    func sendMoney(amount: Decimal, to recipient: String) async throws -> String {
        // WebClient.execute() internally checks circuit breaker
        let data = try client.execute(service: "transfers") {
            try callTransferAPI(amount: amount, to: recipient)
        }
        return parseTransferId(from: data)
    }
}

// ViewModel — doesn't know CircuitBreaker exists
class TransferViewModel: ObservableObject {
    @Published var status: TransferStatus = .idle
    @Published var error: String?

    private let service: TransferService

    enum TransferStatus {
        case idle, loading, success(String), failed
    }

    func send(amount: Decimal, to recipient: String) async {
        status = .loading
        do {
            let id = try await service.sendMoney(amount: amount, to: recipient)
            status = .success(id)
        } catch WebClientError.circuitOpen(let service) {
            error = "Service temporarily unavailable. Please try again in a few minutes."
            status = .failed
        } catch {
            error = "Transfer failed. Please try again."
            status = .failed
        }
    }
}

// View — observes ViewModel
struct TransferView: View {
    @StateObject private var vm = TransferViewModel()
    @State private var amount = ""
```

```
    var body: some View {
        VStack(spacing: 16) {
            TextField("Amount", text: $amount)
                .keyboardType(.decimalPad)

            Button("Send") {
                Task {
                    await vm.send(
                        amount: Decimal(string: amount) ?? 0,
                        to: "recipient-id"
                    )
                }
            }
            .disabled(vm.status == .loading)

            switch vm.status {
            case .idle: EmptyView()
            case .loading: ProgressView()
            case .success(let id): Text("Sent! ID: \(id)")
            case .failed: Text(vm.error ?? "Failed").foregroundColor(.red)
            }
        }
        .padding()
    }
}
```

What to SAY:

> "The CircuitBreaker is an infrastructure concern — it lives inside the networking layer. The ViewModel doesn't know or care about it. It just calls `service.sendMoney()` and handles success or failure. If the circuit is open, the WebClient throws a specific error, and the ViewModel translates that into a user-friendly message: 'Service temporarily unavailable.' This separation means the View stays clean, the ViewModel stays testable, and the circuit breaker logic is reusable across all network calls."

Cache with TTL → SwiftUI Integration

After you code the LRUCache, they ask: "*How would exchange rates flow from this cache to the screen?*"

swift

```
// Repository layer — decides: cache or network?
class RateRepository {
    private let cache: LRUCacheWithTTL<String, Decimal>
    private let api: RateAPIClient

    init(cache: LRUCacheWithTTL<String, Decimal>, api: RateAPIClient) {
        self.cache = cache
        self.api = api
    }

    func getRate(from: String, to: String) async throws -> Decimal {
        let key = "\(from)-\(to)"

        // Check cache first
        if let cached = cache.get(key) {
            return cached  // Cache hit — return immediately
        }

        // Cache miss — fetch from API
        let rate = try await api.fetchRate(from: from, to: to)
        cache.set(key, value: rate)  // Store for next time
        return rate
    }
}

// ViewModel — uses repository, publishes state for the View
class RateViewModel: ObservableObject {
    @Published var displayRate: String = ""
    @Published var isLoading = false
    @Published var lastUpdated: String = ""

    private let repository: RateRepository
    private let formatter: NumberFormatter = {
        let f = NumberFormatter()
        f.numberStyle = .decimal
        f.maximumFractionDigits = 4
        return f
    }()

    func fetchRate(from: String, to: String) async {
        isLoading = true
        defer { isLoading = false }

        do {
            let rate = try await repository.getRate(from: from, to: to)
            displayRate = "1 \(from) = \(formatter.string(for: rate) ?? "") \(to)"
```

```
            lastUpdated = "Updated just now"
        } catch {
            displayRate = "Rate unavailable"
        }
    }
}

// View — simple, declarative, just renders ViewModel state
struct RateDisplayView: View {
    @ObservedObject var vm: RateViewModel
    let fromCurrency: String
    let toCurrency: String

    var body: some View {
        VStack(alignment: .leading, spacing: 8) {
            if vm.isLoading {
                HStack {
                    ProgressView()
                    Text("Fetching rate...")
                        .foregroundColor(.secondary)
                }
            } else {
                Text(vm.displayRate)
                    .font(.title2)
                    .fontWeight(.semibold)

                Text(vm.lastUpdated)
                    .font(.caption)
                    .foregroundColor(.secondary)
            }
        }
        .task {
            await vm.fetchRate(from: fromCurrency, to: toCurrency)
        }
    }
}
```

What to SAY:

> "The cache sits behind a Repository. The ViewModel calls `repository.getRate()` and doesn't know whether the data came from cache or network — that's the Repository's job. The View uses `.task` to trigger the async fetch when it appears. `.task` is the SwiftUI way to launch async work — it automatically cancels when the View disappears, so we don't have orphaned network calls."

Currency Converter → SwiftUI Integration

After you code the converter, they ask: "*How would the transfer form work with this?*"

```swift

```

```swift
// ViewModel that uses YOUR CurrencyConverter + FeeCalculator
class TransferFormViewModel: ObservableObject {
    @Published var sourceAmount: String = ""
    @Published var targetAmount: String = ""
    @Published var feeDisplay: String = ""
    @Published var rateDisplay: String = ""
    @Published var isQuoteExpired = false
    @Published var canSend = false

    private let converter: CurrencyConverter
    private let feeCalculator: FeeCalculator
    private var quoteExpiresAt: Date?

    // Debounce — don't fetch rate on every keystroke
    private var debounceTask: Task<Void, Never>?

    func onAmountChanged(_ newAmount: String) {
        // Cancel previous debounce
        debounceTask?.cancel()

        // Wait 500ms after user stops typing
        debounceTask = Task {
            try? await Task.sleep(nanoseconds: 500_000_000)
            guard !Task.isCancelled else { return }
            await fetchQuote()
        }
    }

    @MainActor
    private func fetchQuote() async {
        guard let amount = Decimal(string: sourceAmount), amount > 0 else {
            targetAmount = ""
            canSend = false
            return
        }

        do {
            let rate = try converter.rateProvider.getRate(from: "USD", to: "EUR")
            let fee = feeCalculator.calculateFee(amount: amount, from: "USD", to: "EUR")
            let converted = try converter.convert(amount: amount - fee, from: "USD", to: "EUR")

            targetAmount = "\(converted)"
            feeDisplay = "Fee: $\(fee)"
            rateDisplay = "1 USD = \(rate) EUR"
            quoteExpiresAt = Date().addingTimeInterval(30)
            isQuoteExpired = false
```

```
            canSend = true
        } catch {
            targetAmount = "Unavailable"
            canSend = false
        }
    }
}

// View
struct TransferFormView: View {
    @StateObject private var vm = TransferFormViewModel()

    var body: some View {
        VStack(spacing: 20) {
            // Source amount input
            VStack(alignment: .leading) {
                Text("You send").font(.caption).foregroundColor(.secondary)
                TextField("0.00", text: $vm.sourceAmount)
                    .keyboardType(.decimalPad)
                    .font(.largeTitle)
                    .onChange(of: vm.sourceAmount) { newValue in
                        vm.onAmountChanged(newValue)
                    }
            }

            // Rate + Fee info
            if !vm.rateDisplay.isEmpty {
                VStack(spacing: 4) {
                    Text(vm.rateDisplay).font(.subheadline)
                    Text(vm.feeDisplay).font(.caption).foregroundColor(.secondary)
                }
                .padding()
                .background(Color(.systemGray6))
                .cornerRadius(8)
            }

            // Target amount display
            VStack(alignment: .leading) {
                Text("They receive").font(.caption).foregroundColor(.secondary)
                Text(vm.targetAmount)
                    .font(.largeTitle)
                    .foregroundColor(.green)
            }

            // Send button
            Button(action: { /* confirm transfer */ }) {
                Text("Send")
```

```
                .frame(maxWidth: .infinity)
                .padding()
                .background(vm.canSend ? Color.green : Color.gray)
                .foregroundColor(.white)
                .cornerRadius(12)
            }
            .disabled(!vm.canSend || vm.isQuoteExpired)
        }
        .padding()
    }
}
```

What to SAY:

> "The key iOS-specific challenge here is debouncing. When the user types '1000', that's four keystrokes — we don't want to fire four API calls. I use a Task-based debounce: cancel the previous task, wait 500ms, then fetch. If the user types another character within 500ms, the old task is cancelled and a new one starts. This is the same pattern you'd use with Combine's `.debounce` operator, but done with structured concurrency.
> The View uses `onChange(of:)` to react to text changes, and the ViewModel handles all the logic. The View is completely dumb — it just renders state."

HTTP Client → SwiftUI Integration

After you code the RetryHTTPClient, they ask: "*How does error handling surface to the user?*"

```swift
```

```swift
// ViewModel translates technical errors into user-facing messages
class DashboardViewModel: ObservableObject {
    @Published var balance: String = ""
    @Published var errorMessage: String?
    @Published var showRetry = false
    @Published var isLoading = false

    private let client: RetryHTTPClient

    func loadBalance() async {
        isLoading = true
        errorMessage = nil
        defer { isLoading = false }

        do {
            struct BalanceResponse: Decodable { let amount: Decimal; let currency: String }
            let response = try await client.fetch(
                BalanceResponse.self,
                from: "https://api.wise.com/v1/balance"
            )
            balance = "\(response.currency) \(response.amount)"
            showRetry = false
        } catch let error as HTTPError {
            switch error {
            case .badStatusCode(401):
                errorMessage = "Session expired. Please log in again."
                showRetry = false  // Don't retry auth errors
            case .badStatusCode(let code) where code >= 500:
                errorMessage = "Something went wrong. Tap to retry."
                showRetry = true   // Server error — retry makes sense
            case .networkFailure:
                errorMessage = "No internet connection."
                showRetry = true
            default:
                errorMessage = "Unable to load balance."
                showRetry = true
            }
        } catch {
            errorMessage = "Something went wrong."
            showRetry = true
        }
    }
}

// View with error handling UI
struct DashboardView: View {
```

```
    @StateObject private var vm = DashboardViewModel()

    var body: some View {
        VStack {
            if vm.isLoading {
                ProgressView()
            } else if let error = vm.errorMessage {
                VStack(spacing: 12) {
                    Image(systemName: "exclamationmark.triangle")
                        .font(.largeTitle)
                        .foregroundColor(.orange)
                    Text(error)
                        .multilineTextAlignment(.center)
                    if vm.showRetry {
                        Button("Try Again") {
                            Task { await vm.loadBalance() }
                        }
                    }
                }
            } else {
                Text(vm.balance)
                    .font(.largeTitle)
                    .fontWeight(.bold)
            }
        }
        .task { await vm.loadBalance() }
    }
}
```

What to SAY:

> "The HTTP client handles retry logic transparently — the ViewModel doesn't know retries are happening. But the ViewModel DOES translate specific error types into appropriate user actions. A 401 means 're-authenticate, don't retry.' A 500 means 'show retry button.' No network means 'tell the user to check their connection.' Each error type maps to a different user experience. This is why typed errors matter — a generic catch-all can't provide this level of UX quality."

Merge Intervals → SwiftUI Integration

This one is less likely to have a SwiftUI follow-up, but if they ask: "*Where would this pattern appear in our app?*"

```swift
```

```swift
// Transfer scheduling — merge overlapping processing windows
struct ProcessingWindow {
    let start: Date
    let end: Date
}

class TransferScheduleViewModel: ObservableObject {
    @Published var availableWindows: [ProcessingWindow] = []
    @Published var nextAvailable: String = ""

    func loadAvailableWindows(for corridor: String) async {
        // Fetch processing windows from API
        let bankWindows = await fetchBankWindows(corridor)    // e.g., [9AM-5PM]
        let wiseWindows = await fetchWiseWindows(corridor)    // e.g., [8AM-10PM]

        // Intersection = when BOTH are available
        let available = intervalIntersection(bankWindows, wiseWindows)

        availableWindows = available
        if let first = available.first {
            nextAvailable = "Next available: \(formatTime(first.start))"
        } else {
            nextAvailable = "No processing windows available today"
        }
    }
}
```

What to SAY:

> "The interval algorithms are useful behind the scenes — the user never sees 'intervals', they see 'your transfer will arrive between 2PM and 5PM.' The algorithm determines those windows by intersecting bank processing hours with Wise's processing capabilities for that corridor."

PART 3: SWIFTUI PATTERNS CHEAT SHEET

The absolute minimum to have in your head during the interview:

STATE MANAGEMENT:

```
┌──────────────────────────────────────────────────────┐
│ @State          → Simple local UI state              │
│ @Binding        → Child modifies parent's state      │
│ @StateObject    → Creates + owns ViewModel           │
│ @ObservedObject → Observes external ViewModel        │
│ @Published      → Inside ViewModel, triggers UI      │
│ @EnvironmentObject → App-wide shared state           │
└──────────────────────────────────────────────────────┘
```

ASYNC IN SWIFTUI:

```
┌──────────────────────────────────────────────────────┐
│ .task { }       → Launch async on appear             │
│                   Auto-cancels on disappear          │
│ .task(id: x) {} → Re-launch when x changes           │
│ .refreshable {} → Pull-to-refresh (async)            │
│ Task { }        → Fire async from button action      │
└──────────────────────────────────────────────────────┘
```

VIEW LIFECYCLE:

```
┌──────────────────────────────────────────────────────┐
│ .onAppear { }   → View appeared (sync only)          │
│ .onDisappear {} → View left screen                   │
│ .task { }       → Like onAppear but async            │
│ .onChange(of:) {} → React to state changes           │
└──────────────────────────────────────────────────────┘
```

NAVIGATION (SwiftUI 4+):

```
┌──────────────────────────────────────────────────────┐
│ NavigationStack { }                                  │
│ .navigationDestination(for: Type.self) { }           │
│ .sheet(isPresented:) { }                             │
│ .alert(isPresented:) { }                             │
└──────────────────────────────────────────────────────┘
```

PART 4: THE 10-SECOND ANSWERS

If any SwiftUI topic comes up and you need a quick answer:

Topic	10-Second Answer
@StateObject vs @ObservedObject	"StateObject creates and OWNS the ViewModel — use when the View creates it. ObservedObject observes but doesn't own — use when it's passed from a parent."
Why structs for Views?	"Lightweight, no ARC overhead, value semantics. SwiftUI recreates View structs constantly — cheap to create, diff, and discard."
.task vs .onAppear	".task supports async/await and auto-cancels when the View disappears. onAppear is synchronous only."
Combine in SwiftUI	"Used for reactive streams — debouncing user input, combining multiple data sources, receiving on main thread. @Published is actually a Combine publisher."
How SwiftUI updates	"When a @Published property changes, SwiftUI diffs the View body, finds what changed, and re-renders only those parts. We don't manually update the UI."
List performance	"Use identifiable items, avoid complex computations in body, use .id() for stable identity, and lazy loading with LazyVStack for large datasets."
Error handling in UI	"ViewModel exposes @Published error: String? — View shows alert or inline error based on this. Never throw from View body."
Dependency Injection	"Pass dependencies through ViewModel init. For app-wide deps, use @EnvironmentObject. Never create services inside Views."
Testing SwiftUI	"Don't test Views directly — test ViewModels. The ViewModel is plain Swift with no SwiftUI dependency, fully testable with XCTest."
Previews	"Use #Preview with mock data. Create a PreviewMockService that returns hardcoded data so previews don't hit real APIs."

PART 5: HOW TO STUDY THIS (30 minutes total)

Do NOT code any of these SwiftUI views on HackerRank. The pair programming problem will be a logic/architecture problem (Circuit Breaker, Cache, etc.), not a SwiftUI screen.

Instead:

15 minutes: Read Part 1 (discussion answers) out loud. These are the verbal responses you'll give in the first 5 minutes.

10 minutes: For each of the 5 problems you're practicing, read the "What to SAY" section for its SwiftUI integration. Don't memorize code — memorize the CONNECTION between your solution and the app.

5 minutes: Glance at the cheat sheet in Part 3 and the 10-second answers in Part 4. These are fallback answers if they ask something unexpected.

That's it. Then go back to practicing the 5 actual coding problems.