

# WISE ROUND 2: PAIR PROGRAMMING — MOBILE ENGINEERING MANAGER

## What's Different When You're Interviewing for a Manager Role

60 min | Two Engineers (likely Engineering Leads) | HackerRank | Zoom

---

### HOW THIS ROUND IS DIFFERENT FOR A MANAGER

#### From the Candidate Pack:

"Discussion on generic computer science problems and **specific domain related questions about iOS/Android** with two engineers."

#### From Glassdoor (Manager-level candidates):

"Had an initial HR screening followed by a pair programming round with **2 head of engineering leads**."

#### From Wise's EM Tips Page:

"We're looking to understand your approach on high-level design choices, documentation and how you ensure **best practice**."

#### From Blind:

"It was basically a **code fluency** exercise. Not algorithms focused, you just need to be able to **write clean code**."

#### What this means for you:

IC Interview	Manager Interview
"Can you solve this algorithm?"	"Can you still code AND think architecturally?"
Correctness is king	<b>Code quality + reasoning</b> matters more
Just solve the problem	"How would you review this code?" "How would you test this?"
Speed matters	<b>Communication and collaboration</b> matter more
Interviewers are engineers	Interviewers are <b>engineering leads</b>
They test DS/Algo depth	They test <b>breadth + leadership thinking</b>

**The bottom line:** As a Manager candidate, they're not looking for LeetCode speed. They want to see that you can:

1. Write clean, production-quality code (not just hack a solution)
  2. Think about testability, extensibility, and maintenance
  3. Discuss trade-offs and justify decisions
  4. Collaborate with the interviewers like a peer
  5. Show iOS/mobile domain expertise naturally through your coding choices
- 

## THE 60-MINUTE BREAKDOWN (Manager-Level)

0:00 - 0:05 Introductions

→ "Tell us about yourself" (30 seconds, focus on iOS + leadership)

0:05 - 0:10 iOS/Mobile domain discussion

→ Questions about your experience, frameworks, opinions

→ "What's your view on SwiftUI vs UIKit?"

→ "How do you approach modularisation?"

→ "What patterns do you use for networking?"

0:10 - 0:50 Coding task on HackerRank (the main event)

→ They give you a problem

→ You solve it collaboratively

→ They add requirements (follow-ups)

→ They discuss improvements

0:50 - 0:55 "How would you test this?" / "What would you change?"

→ This is the MANAGER SIGNAL question

0:55 - 1:00 Your questions for them

---

## PHASE 1: iOS/MOBILE DOMAIN DISCUSSION (5-10 min)

They'll ask opinion-based questions before coding. For a Manager, these carry more weight than for an IC.

### Q: "What's your view on SwiftUI vs UIKit?"

"SwiftUI is the clear future — Apple's investment is entirely there, and I know Wise has already migrated from UIKit to SwiftUI with Combine. For a new feature, I'd default to SwiftUI. But I'd keep UIKit knowledge on the team because there are still edge cases — complex custom layouts, certain UIKit-only APIs, and the reality that some teams will encounter UIKit in legacy code during refactoring.

As a manager, the bigger question isn't SwiftUI vs UIKit — it's how do you manage the migration without disrupting feature delivery. At PayPal, I led my team's adoption of modern concurrency patterns alongside

feature work. You can't stop shipping features for 6 months to rewrite. You incrementally migrate, starting with new features in SwiftUI and wrapping legacy UIKit screens with UIHostingController."

**Q: "How do you approach modularisation?"**

"I'd modularise by feature, not by layer. A 'TransferModule' contains its views, view models, repositories, and models — everything needed for the transfer feature. It depends on shared modules like 'NetworkKit' and 'DesignSystem' but not on other feature modules.

The benefits are clear: teams can work on features independently (which aligns with Wise's autonomous squad model), build times improve because you only recompile changed modules, and you can test each module in isolation.

At PayPal, the iOS app is a monolith, and I've seen the problems that causes — long build times, merge conflicts across teams, and difficulty isolating bugs. Modularisation solves all of these."

**Q: "What patterns do you use for networking?"**

"Protocol-based API client with async/await. The networking layer depends on a `NetworkSession` protocol — in production it's URLSession, in tests it's a mock. The API client handles request construction, authentication headers, response validation, and JSON decoding.

Above that, a Repository layer decides whether to return cached data or make a network call. The ViewModel calls the Repository and never knows about URLSession.

For a fintech app, I'd also add certificate pinning, request retry with exponential backoff, and idempotency keys for mutating requests. These aren't optional for an app that moves money."

**Q: "What iOS-specific challenges do you think about as a lead?"**

"App launch time — every second matters for daily-use fintech apps. Memory management — especially with large transaction lists. Background refresh — keeping balances and transfer statuses updated. Push notification handling — deep linking to the right screen. And most importantly, the release cycle — you can't hotfix like web. Every release needs to be solid, which is why I invest in CI/CD, automated testing, and feature flags."

---

## **PHASE 2: THE CODING TASK (30-40 min)**

### **What Managers Actually Get Asked**

Based on all confirmed data points, manager-level pair programming at Wise tends toward **practical, system-like problems** rather than pure algorithm questions:

Confirmed Question	Type	Difficulty
<b>Circuit Breaker</b>	System design + code	Medium
<b>Simple Cache</b> → extend with TTL	Build + extend	Medium
<b>Currency Converter</b> → add fees/expiry	OOP + extend	Medium
<b>Add functionality to existing code</b>	Extend existing	Medium
<b>HTTP request handling</b>	Practical	Medium
<b>Sort 2 lists of intervals</b>	Algorithm	Medium
<b>Combination Sum</b>	Algorithm	Medium

For a Manager, the **Circuit Breaker** and **Currency Converter** are the most likely because they test:

- OOP design (protocols, classes, separation of concerns)
- Incremental development (start simple, add features)
- Error handling and edge cases
- Testability thinking
- Trade-off discussion

## HOW TO CODE LIKE A MANAGER (not an IC)

Here's the exact same problem solved two ways. The first is how an IC might solve it. The second is how a Manager should solve it:

**Problem:** "Implement a currency converter"

### ✗ IC Approach (functional but not leadership-level):

```
swift

func convert(amount: Double, from: String, to: String, rates: [String: Double]) -> Double {
    let key = "\(from)-\(to)"
    guard let rate = rates[key] else { return -1 }
    return amount * rate
}
```

Works. Passes tests. But shows nothing about how you think about real systems.

### ✓ Manager Approach (shows architectural thinking):

```
swift
```

```

// Step 1: Use Decimal for financial calculations
// "I'm using Decimal instead of Double because floating-point
// arithmetic has precision issues that are unacceptable for money."

// Step 2: Protocol for testability
// "I'm defining a protocol so we can inject a mock rate provider in tests."
protocol RateProvider {
    func getRate(from: String, to: String) throws -> Decimal
}

// Step 3: Typed errors
// "I'm creating specific error types so the caller knows exactly what went wrong."
enum ConversionError: Error {
    case unsupportedCurrency(String)
    case rateNotAvailable(from: String, to: String)
    case invalidAmount
}

// Step 4: Clean implementation
struct CurrencyConverter {
    private let rateProvider: RateProvider

    init(rateProvider: RateProvider) {
        self.rateProvider = rateProvider
    }

    func convert(amount: Decimal, from: String, to: String) throws -> Decimal {
        guard amount > 0 else { throw ConversionError.invalidAmount }
        guard from != to else { return amount }

        let rate = try rateProvider.getRate(from: from, to: to)
        return amount * rate
    }
}

// Step 5: Concrete implementation
class InMemoryRateProvider: RateProvider {
    private var rates: [String: Decimal] = [:]

    func addRate(from: String, to: String, rate: Decimal) {
        rates["\(from)-\(to)"] = rate
        rates["\(to)-\(from)"] = 1 / rate
    }

    func getRate(from: String, to: String) throws -> Decimal {
        guard let rate = rates["\(from)-\(to)"] else {

```

```
        throw ConversionError.rateNotAvailable(from: from, to: to)
    }
    return rate
}
}
```

### What to SAY while coding this:

*Writing the protocol:* "I'm starting with a protocol because this is how I'd structure it for a team — anyone can implement `RateProvider`, and we can test the converter without real network calls."

*Writing the error enum:* "In fintech, generic errors aren't acceptable. If a conversion fails, the user needs to know whether it's an unsupported currency, an expired rate, or invalid input. Typed errors let us show the right error message."

*Using Decimal:* "I'm using Decimal because this is financial — Double would introduce precision errors."

*guard amount > 0:* "I always validate inputs first. In a code review, I'd reject any financial function that doesn't check for negative or zero amounts."

*guard from != to:* "Edge case — converting USD to USD should just return the amount, not hit the rate provider."

### WHEN THEY ADD REQUIREMENTS:

**Interviewer:** "Now add fee calculation."

### Manager approach:

swift

```

// "I'd add this as a separate component, not inline in the converter.
// Single Responsibility — the converter converts, the fee calculator calculates fees."

protocol FeeCalculator {
    func calculateFee(amount: Decimal, from: String, to: String) -> Decimal
}

struct PercentageFeeCalculator: FeeCalculator {
    private let feePercentage: Decimal
    private let minimumFee: Decimal

    init(percentage: Decimal = Decimal(string: "0.005")!,
         minimum: Decimal = Decimal(string: "1.00")!) {
        self.feePercentage = percentage
        self.minimumFee = minimum
    }

    func calculateFee(amount: Decimal, from: String, to: String) -> Decimal {
        max(amount * feePercentage, minimumFee)
    }
}

// Extend converter to use fees:
struct TransferQuoteService {
    private let converter: CurrencyConverter
    private let feeCalculator: FeeCalculator

    struct Quote {
        let sourceAmount: Decimal
        let fee: Decimal
        let rate: Decimal
        let targetAmount: Decimal
    }

    func getQuote(amount: Decimal, from: String, to: String) throws -> Quote {
        let fee = feeCalculator.calculateFee(amount: amount, from: from, to: to)
        let amountAfterFee = amount - fee
        let rate = try converter.rateProvider.getRate(from: from, to: to)
        let targetAmount = try converter.convert(amount: amountAfterFee, from: from, to: to)

        return Quote(sourceAmount: amount, fee: fee, rate: rate, targetAmount: targetAmount)
    }
}

```

**What to SAY:**

"I'm not putting fee logic inside the converter. That violates Single Responsibility. If we later need different fee structures per payment method — bank transfer vs card — we just create different FeeCalculator implementations. Open/Closed Principle."

**Interviewer:** "Now what about rate expiry?"

```
swift

// "Rates change frequently. I'd wrap rates with a timestamp and TTL."

struct TimedRate {
    let rate: Decimal
    let fetchedAt: Date
    let ttlSeconds: TimeInterval

    var isExpired: Bool { Date().timeIntervalSince(fetchedAt) > ttlSeconds }
}

class ExpiringRateProvider: RateProvider {
    private var rates: [String: TimedRate] = [:]
    private let ttl: TimeInterval

    init(ttlSeconds: TimeInterval = 30) { self.ttl = ttlSeconds }

    func addRate(from: String, to: String, rate: Decimal) {
        let key = "\\(from)-\\(to)"
        rates[key] = TimedRate(rate: rate, fetchedAt: Date(), ttlSeconds: ttl)
    }

    func getRate(from: String, to: String) throws -> Decimal {
        let key = "\\(from)-\\(to)"
        guard let timedRate = rates[key] else {
            throw ConversionError.rateNotAvailable(from: from, to: to)
        }
        guard !timedRate.isExpired else {
            throw ConversionError.rateExpired
        }
        return timedRate.rate
    }
}
```

### What to SAY:

"At Wise, rates have a ~30-second validity window. Showing an expired rate and honoring it on confirmation means Wise absorbs the FX risk. This TTL check ensures we never convert at a stale rate. This is exactly the kind of problem I work on at PayPal with BFX."

## PHASE 3: THE "MANAGER SIGNAL" QUESTIONS (5-10 min)

After you finish coding, they'll ask questions that distinguish managers from ICs:

### Q: "How would you test this code?"

"Three levels:

**Unit tests** for the converter itself — inject a `MockRateProvider` that returns known rates, verify conversion math. Test edge cases: zero amount, negative amount, same currency, unsupported currency, expired rate.

**Unit tests** for the fee calculator — verify percentage calculation, minimum fee floor, different amounts.

**Integration test** for the `QuoteService` — verify the full flow: amount → fee deduction → conversion → correct target amount. Use a `MockRateProvider` so we're not testing the network.

For the expiring rate, I'd inject a `MockTimeProvider` so tests can control time instead of waiting real seconds."

swift

// Quick test example I'd mention:

```
func testConvert_validAmount_returnsCorrectResult() throws {
    let mock = InMemoryRateProvider()
    mock.addRate(from: "USD", to: "EUR", rate: Decimal(string: "0.92")!)
    let converter = CurrencyConverter(rateProvider: mock)

    let result = try converter.convert(amount: 1000, from: "USD", to: "EUR")
    XCTAssertEqual(result, Decimal(string: "920")!)
}

func testConvert_zeroAmount_throwsError() {
    let converter = CurrencyConverter(rateProvider: InMemoryRateProvider())
    XCTAssertThrowsError(try converter.convert(amount: 0, from: "USD", to: "EUR"))
}
```

### Q: "What would you improve if you had more time?"

"Several things:

**Thread safety** — the rate storage isn't thread-safe. Multiple screens fetching rates simultaneously could cause a data race. I'd make the rate provider an actor, or use a serial dispatch queue for synchronization.

**Indirect conversion** — right now if we only have USD→EUR and EUR→GBP rates, we can't convert USD→GBP. I'd add graph-based path finding using BFS to chain conversions through intermediate currencies. But I'd flag to the team that chained conversions compound rounding errors, so we should limit to one intermediary.

**Monitoring** — in production, I'd add logging for expired rate hits (tells us if the TTL is too short), conversion error rates by currency pair (tells us if specific corridors have issues), and latency metrics on rate fetching.

**Feature flags** — I'd wrap this behind a feature flag for gradual rollout, exactly like I do with BFX at PayPal."

**Q: "If a junior on your team wrote the initial version (the IC version), how would you review it?"**

"I'd acknowledge what works — the logic is correct and it solves the immediate problem. Then I'd give constructive feedback:

1. 'Let's use Decimal instead of Double — here's why...' (explain the 0.1 + 0.2 problem)
2. 'Can we make this testable? If we extract a protocol for the rate source, we can test the converter without network calls.'
3. 'What happens if amount is negative? Let's add input validation.'
4. 'Let's create typed errors instead of returning -1 — the caller needs to know WHY it failed.'

I wouldn't rewrite their code. I'd pair with them to improve it, explaining the reasoning at each step. That's how you grow engineers — through code review as a teaching tool."

---

## **THE CIRCUIT BREAKER (MOST LIKELY MANAGER-LEVEL QUESTION)**

This is confirmed on Glassdoor (Oct 2025, Dec 2024 — got offer). Here's the manager-quality approach:

### **The Prompt:**

"We have a web client that makes requests to Service B and Service C. If either service fails 3 times within 10 minutes, stop making requests for 5 minutes. After 5 minutes, try again."

### **Manager Approach:**

swift

```

// STEP 1: Define the state machine clearly
// "A circuit breaker is a state machine with three states."
enum CircuitState {
    case closed    // Normal — requests pass through
    case open      // Tripped — requests blocked
    case halfOpen  // Testing — allow one request to check recovery
}

// STEP 2: Protocol for testability
// "I'm making time injectable so tests don't need to wait real minutes."
protocol Clock {
    func now() -> Date
}

struct SystemClock: Clock { func now() -> Date { Date() } }

// STEP 3: The circuit breaker
class CircuitBreaker {
    private let failureThreshold: Int
    private let failureWindow: TimeInterval
    private let recoveryTimeout: TimeInterval
    private let clock: Clock

    private(set) var state: CircuitState = .closed
    private var failureTimestamps: [Date] = []
    private var openedAt: Date?

    init(failureThreshold: Int = 3,
         failureWindowSeconds: TimeInterval = 600,
         recoveryTimeoutSeconds: TimeInterval = 300,
         clock: Clock = SystemClock()) {
        self.failureThreshold = failureThreshold
        self.failureWindow = failureWindowSeconds
        self.recoveryTimeout = recoveryTimeoutSeconds
        self.clock = clock
    }

    func canExecute() -> Bool {
        let now = clock.now()
        switch state {
        case .closed:
            return true
        case .open:
            guard let openedAt = openedAt else { return false }
            if now.timeIntervalSince(openedAt) >= recoveryTimeout {
                state = .halfOpen
                return true // Allow one test request
            }
        }
    }
}

```

```

    }

    return false
case .halfOpen:
    return true
}
}

func recordSuccess() {
    if state == .halfOpen {
        state = .closed
        failureTimestamps.removeAll()
        openedAt = nil
    }
}

func recordFailure() {
    let now = clock.now()
    switch state {
    case .closed:
        failureTimestamps.append(now)
        // Remove failures outside the window
        failureTimestamps.removeAll { now.timeIntervalSince($0) > failureWindow }
        if failureTimestamps.count >= failureThreshold {
            state = .open
            openedAt = now
        }
    case .halfOpen:
        state = .open
        openedAt = now
    case .open:
        break
    }
}
}

// STEP 4: WebClient with per-service circuit breakers
enum WebClientError: Error {
    case circuitOpen(service: String)
    case requestFailed(Error)
}

class WebClient {
    private var breakers: [String: CircuitBreaker] = [:]

    private func breaker(for service: String) -> CircuitBreaker {
        if let cb = breakers[service] { return cb }
        let cb = CircuitBreaker()
    }
}

```

```

breakers[service] = cb
return cb
}

func execute(service: String, request: () throws -> Data) throws -> Data {
    let cb = breaker(for: service)

    guard cb.canExecute() else {
        throw WebClientError.circuitOpen(service: service)
    }

    do {
        let result = try request()
        cb.recordSuccess()
        return result
    } catch {
        cb.recordFailure()
        throw WebClientError.requestFailed(error)
    }
}
}

```

### What to SAY while coding:

*Defining the enum:* "Circuit breaker is fundamentally a state machine. Making the states an enum makes invalid states unrepresentable."

*Injecting Clock:* "I'm injecting a Clock protocol so tests can control time. Without this, you'd need to sleep for 5 real minutes in your test suite, which is unacceptable."

*Sliding window:* "I use timestamps instead of a simple counter because a counter never resets. If we had 2 failures 30 minutes ago and 1 failure now, a counter would say 3 and open the circuit. The sliding window correctly ignores the old failures."

*Per-service breakers:* "Each service gets its own breaker so Service B's failures don't block requests to Service C."

### When they ask "How would you test this?":

swift

```

func testCircuit_opensAfterThreeFailures() {
    let mockClock = MockClock()
    let cb = CircuitBreaker(clock: mockClock)

    cb.recordFailure()
    mockClock.advance(by: 60)
    cb.recordFailure()
    mockClock.advance(by: 60)
    cb.recordFailure()

    XCTAssertEqual(cb.state, .open)
    XCTAssertFalse(cb.canExecute())
}

func testCircuit_recoversAfterTimeout() {
    let mockClock = MockClock()
    let cb = CircuitBreaker(recoveryTimeoutSeconds: 300, clock: mockClock)

    // Open it
    cb.recordFailure(); cb.recordFailure(); cb.recordFailure()
    XCTAssertEqual(cb.state, .open)

    // Advance past recovery timeout
    mockClock.advance(by: 301)
    XCTAssertTrue(cb.canExecute()) // Should be halfOpen now

    cb.recordSuccess()
    XCTAssertEqual(cb.state, .closed) // Recovered!
}

```

## MANAGER-LEVEL BEHAVIORAL SIGNALS DURING CODING

These are the things Engineering Leads watch for that separate managers from ICs:

Signal	How to Show It
Names things well	<code>CircuitState</code> , not <code>State</code> . <code>failureThreshold</code> , not <code>threshold</code> . <code>openedAt</code> , not <code>time</code> .
Thinks about the team	"If a junior reads this code in 6 months, will they understand it?"
Considers production	"In production, I'd add logging when the circuit opens — the ops team needs to know."
Tests naturally	"Let me write a quick test to verify this works" — don't wait to be asked
Discusses trade-offs	"I chose sliding window over counter because..."
Thinks about monitoring	"I'd want a dashboard showing circuit state per service."
References real experience	"At PayPal, we had a similar pattern with BFX rollouts..."
Asks good questions	"Should the circuit breaker be shared across the app, or per-screen?"
Considers thread safety	"If multiple threads call this simultaneously, we have a race condition."
Doesn't over-engineer	Start simple, add complexity when asked

## TOP 5 QUESTIONS TO PRACTICE ON HACKERRANK

Do these in order. Time yourself: 40 minutes each, Swift, no Xcode.

#	Problem	Why It's Most Likely	What Managers Add
1	<b>Circuit Breaker</b> (custom)	Confirmed Glassdoor (got offer)	Injectable time, per-service, testability
2	<b>Currency Converter</b> → add fees → add TTL	Confirmed Glassdoor (multiple)	Decimal, protocols, SOLID principles
3	<b>Cache with TTL</b> → extend to LRU	Confirmed Mar 2025	Thread safety discussion, eviction strategy
4	<b>HTTP Client</b> → add retry → add circuit breaker	Confirmed Reddit	Protocol-based, async/await, error handling
5	<b>Merge Intervals</b> (LC 56) → add Wise context	Confirmed Wise blog	Clean code, edge cases, complexity analysis

For each problem, practice saying out loud:

1. "I'm choosing this approach because..."

2. "The trade-off here is..."
  3. "I'd test this by..."
  4. "In production, I'd also add..."
  5. "If a junior wrote this, I'd suggest..."
- 

## WHAT TO BRING TO THE INTERVIEW

- ☐ HackerRank account (sign up, test Swift works)
  - ☐ Quiet room, stable internet, good mic
  - ☐ Water nearby (you'll be talking a lot)
  - ☐ Notepad for jotting down the problem before coding
  - ☐ Mental checklist: Decimal for money, protocols for DI, guard for validation, typed errors
- 

## THE 30-SECOND RULE

For every coding decision, if you can't explain WHY in 30 seconds, you shouldn't make that decision. Practice:

■ "I used Decimal because Double has floating-point precision issues with money." (5 seconds)

■ "I extracted a protocol because this lets us inject mocks for testing and swap implementations without changing the caller." (8 seconds)

■ "I'm using a sliding window for failures because a simple counter would never reset — old failures from hours ago shouldn't count." (10 seconds)

■ "I made the clock injectable because testing time-dependent code without this requires sleeping in tests, which makes the suite slow and flaky." (10 seconds)

If you can do this for every line of code you write, you'll come across as a senior engineering leader, not just a coder.