ChatGPT

Swift and iOS Development: Best Practices and Common Code Smells

Introduction

Writing clean, efficient Swift code is crucial for maintainable and high-performance iOS apps. This guide highlights **best practices** to follow and **bad practices (code smells)** to avoid in a SwiftUI iOS app. By adhering to these guidelines, your development team can prevent "stinky" code and deliver robust code to production.

Avoid Busy Waiting and Polling Loops

One glaring anti-pattern is using **polling loops** (busy-waiting) to wait for a condition. For example:

```
//   Bad Practice: Polling loop to wait for a condition
while !cameraService.isSessionRunning {
    try? await Task.sleep(for: .milliseconds(Int(pollInterval * 1000)))
    // ... other checks or setup
}
```

Such code continuously checks a condition with a delay, which is inefficient. Even though `Task.sleep` yields the thread, the loop ties up a system thread doing nothing most of the time, wasting CPU cycles and battery [1]. As one expert put it, *"Firstly, don't [use polling] if you can [avoid it]."* If you must poll, **use a scheduled timer or appropriate async mechanism instead** [2]. For instance, you could use a `Timer` or `DispatchSourceTimer` for periodic checks rather than a tight loop [2]. Better yet, prefer an **event-driven approach**: use delegation, Combine publishers, or key-value observations to be notified when a condition changes (e.g. observe `isSessionRunning` via KVO or a callback) instead of actively waiting. This ensures the app remains responsive and efficient.

Never Block the Main Thread

In iOS, the main thread handles all UI updates. **Blocking the main thread** (e.g. performing heavy computation or long waits on it) will freeze the UI and degrade user experience. As a rule: *"Never block the main thread — keep UI work smooth."* Use background queues or async tasks for heavy work, and always update UI on the main thread [3]. For example, perform network requests or data processing on a global background `DispatchQueue` or using Swift's async/await concurrency, then switch back to `DispatchQueue.main` for UI updates. By offloading work appropriately, you prevent the dreaded frozen interface and keep your app responsive [3].

Embrace Swift Concurrency (Async/Await)

Swift's modern concurrency (async/await) should be used in place of old-fashioned completion handlers or manual thread sleeps. Async/await leads to more readable, maintainable code for asynchronous tasks [4]. For instance, instead of nesting completion closures (callback hell), use `async func` and `await` the results. This not only clarifies the flow but also avoids mistakes like the polling loop above. **However, use async/await correctly** – avoid designing asynchronous loops that run indefinitely without awaiting meaningful work. If you need a continuous loop (e.g. a periodic task), incorporate `Task.isCancelled` checks and `Task.sleep` delays to yield control, and ensure there's a condition to break out (or a timeout) to prevent runaway tasks. The key is to leverage Swift concurrency's primitives (like structured concurrency, actors, `AsyncSequence`, etc.) rather than inventing low-level loops. This yields safer and more efficient async code.

Safe Optional Handling (Avoid Force-Unwrapping)

Swift's optionals help prevent runtime crashes, but only if used properly. **Never force-unwrap an optional with `!` unless you are absolutely sure it's non-nil**. Force-unwrapping a nil optional will crash your app. For example:

```
let user: User? = fetchUser()
//   Bad: Dangerous force unwrap – will crash if user is nil
let name = user!.name
```

This is a *bad practice* [5]. Instead, use optional binding (`if let` or `guard let`) or nil-coalescing. For example:

```
//   Good: Safe optional handling
if let user = user {
    let name = user.name
    print("Hello, \(name)")
} else {
    print("Hello, Guest")
}
// or using nil-coalescing:
let name = user?.name ?? "Unknown User"
```

By unwrapping safely, you handle the nil case explicitly and avoid crashes [6]. The same goes for implicitly unwrapped optionals (`!` in type) – use them sparingly. **Bottom line:** treat optionals with care and never assume they're non-nil without checks.

Proper Error Handling (Don’t Ignore or Force Errors)

Swift encourages robust error handling with `do-catch` and `throws`. **Avoid using** `try!`, which forcefully ignores error propagation. A `try!` (force try) will crash the app if an error actually occurs, and it bypasses the opportunity to handle the error gracefully [7]. As a code smell example:

```
func loadData() throws -> Data { ... }
func process() {
    let data = try! loadData()  //   Bad: force-try, will crash on error
    // ... use data
}
```

This is risky and makes assumptions that the throwing function will never fail [7]. Instead, **use** `do { try ... } catch { ... }` **to handle errors** or propagate the error to callers. For example:

```
func process() {
    do {
        let data = try loadData()  //   Proper error handling
        // ... use data
    } catch {
        // handle or log the error appropriately
        print("Failed to load data: \(error)")
    }
}
```

This way, you either recover from the error or at least fail gracefully [8] [9]. Similarly, avoid blanket `try?` without handling the result – using `try?` will convert errors to nil and might hide failures. If you use `try?`, be sure to check the result isn’t nil and handle the failure path. In summary, **never ignore errors**; handle them or propagate them using Swift’s error handling mechanisms so that issues don’t go unnoticed in production.

Memory Management: Avoid Retain Cycles in Closures

Be mindful of **strong reference cycles** (retain cycles), especially when using closures that capture `self`. In SwiftUI and iOS development, it’s common to pass closures to asynchronous operations (network responses, timers, etc.). If those closures retain `self` strongly, your View or ViewModel might never deallocate. **Always use** `[weak self]` **(or** `[unowned self]` **when appropriate) in closure capture lists** when `self` is referenced within the closure and the closure is stored or executed asynchronously [10] [11]. For example:

```
Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true) { [weak self] _ in
    self?.updateUI()
}
```

Using `[weak self]` ensures the timer's closure only captures `self` weakly, preventing a retain cycle that would keep `self` alive forever [12]. In contrast, forgetting `[weak self]` (capturing a strong self) is a **bad practice** that can lead to memory leaks:

```
Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true) { _ in
    self.updateUI()  //   Strong capture of self, causes retain cycle
}
```

This strongly retains `self` inside the closure (the timer keeps the closure alive, the closure keeps the view/controller alive), causing a leak [11]. **Rule of thumb:** Use `[weak self]` in escaping closures; if you are sure `self` will outlive the closure, you could use `[unowned self]` [13]. This practice keeps your memory usage in check and avoids crashes due to deallocated objects being accessed.

SwiftUI State Management Best Practices

In a SwiftUI app, how you manage state and data flow greatly affects code quality. Follow these guidelines for SwiftUI-specific good practices:

- **Use the Correct Property Wrappers:** For view model objects (reference types) owned by a SwiftUI view, use `@StateObject` instead of `@ObservedObject`. `@StateObject` ensures the object is created only once and not reinitialized on every re-render [14]. Use `@ObservedObject` only for objects passed in from elsewhere. Misusing these can cause unexpected multiple initializations or updates.
- **Prefer `@State` and `@Binding` for Value Types:** Use `@State` for simple value types owned by the view, and pass bindings to child views that need to mutate parent state. This keeps a single source of truth. Avoid anti-patterns like storing view state in a global or using singletons for state.
- **Avoid Massive Views:** SwiftUI views are structs, but they can still become monolithic and hard to maintain if too large. Break out subviews for independent UI pieces, especially ones that don't need to reload together. For example, if part of the UI is expensive to redraw, move it to its own `View` that takes only the necessary state. This prevents unnecessary re-renders of complex subviews when unrelated state changes [15] [16]. As shown in one example, extracting a subview prevented an expensive animation view from redrawing on each counter update [17] [18].
- **Use Lazy Containers for Large Lists:** If you display long lists or grids, use SwiftUI's `LazyVStack`/`LazyHStack` or lazy grids. A non-lazy container loads all items at once (bad for performance). Lazy containers only create views as needed. For instance, a `ScrollView` with a regular `VStack` of 1000 items will instantiate all 1000 subviews immediately (💣), whereas `LazyVStack` will load on demand (much better for memory usage) [19] [20].
- **Avoid `AnyView` Unless Unavoidable:** Type-erased `AnyView` can hide view type information and hurt performance. Prefer using conditional views with SwiftUI's built-in result builders or view

composition instead of wrapping things in `AnyView` [21] [22]. This keeps SwiftUI's type system working for you and often improves rendering speed.
- **Use Environment and Observables Appropriately:** Take advantage of `@Environment` for globally needed objects (e.g., app settings) and `ObservableObject` with `@Published` for data models. This ties into using SwiftUI's data flow instead of imperatively updating UI. For example, if `cameraService` is an `ObservableObject` with a `@Published isSessionRunning`, your UI or view model could simply listen to that publisher to react when it becomes true, rather than polling. Leveraging Combine or SwiftUI's `.onReceive` and `.task(id:)` modifiers can make your code reactive and clear.

By following these SwiftUI patterns, you avoid common pitfalls like redundant updates, poor performance, or unpredictable state.

Other Clean Code Practices

In addition to the major areas above, keep the codebase clean with these practices:

- **Meaningful Naming and Clarity:** Choose clear, descriptive names for classes, functions, and variables. Avoid abbreviations that aren't obvious. Code is read more often than written, so optimize for readability.
- **Small Functions & Single Responsibility:** Break down large functions into smaller ones that each do one thing. This makes code easier to test and reuse. For example, a view model method that configures and starts the camera session should perhaps be split into `configureSession()` and `startSession()` rather than one huge function.
- **Remove Dead or Debug Code:** Don't leave print statements or commented-out blocks in production code. These clutter the codebase and can even slow down the app (multiple stray `print` calls can degrade performance). If you need logging, use a dedicated logging utility that can be turned off in release builds. Otherwise, *"The `print` command is useful, but easy to forget to remove… you'll end up with a console flooded with messages that become just noise."* [23]. Similarly, eliminate commented-out old code – if it's not needed, delete it to reduce confusion [24]. Version control will preserve history if you ever need it back.
- **Consistent Code Style:** Follow a consistent style guide for braces, spacing, naming conventions, etc. Consistency makes collaboration easier. You can adopt an established style like [GitHub's Swift Style Guide] or Apple's Swift API Design Guidelines. Tools like **SwiftLint** can automate this – SwiftLint will warn or even forbid certain bad patterns and style violations, helping keep code "unsmelly" and organized [25]. For example, you can enable SwiftLint rules to catch force-unwrapping, long methods, or other code smells so they're flagged during development.
- **Modularity and Reuse:** Identify repeated code and abstract it. If you notice the same code snippet in multiple places, consider refactoring into a helper function or extension. This follows the DRY (Don't Repeat Yourself) principle and makes future changes easier (one fix in one place, not many).
- **Adhere to SOLID Principles (When Applicable):** While a full discussion is beyond scope, keep in mind principles like single-responsibility, open-closed, etc., especially for bigger projects. For instance, an MVC or MVVM architecture can help organize code: your SwiftUI Views handle UI, ViewModels handle logic and state, and so on. Avoid huge classes or structs that do everything.

Conclusion and Next Steps

By enforcing these good practices and avoiding the bad ones, your team can significantly improve code quality and app stability. Always **review code for these common smells** during code reviews. If something "feels off" (like a busy waiting loop or a force-unwrap), it likely warrants refactoring – remember that a *code smell is a surface indication of a deeper problem* [26]. Encourage the use of Swift's modern features (optionals, async/await, strong type system) as they are designed to make code safer and cleaner when used properly.

Finally, consider using automated tools (linters, formatters, static analysis) in your development pipeline to catch issues early. SwiftLint, SonarQube, or Xcode's built-in analyze tools can spot many of the bad practices mentioned (force unwrapping, force tries, unused code, etc.) and ensure they don't slip into production. As a team, agree on a standard and hold each other accountable to follow it. Writing clean code is a habit – with this guide and vigilant practice, **your dev team should never produce code that stinks** again!

References and Sources

- Halil Özel, *"Swift Best Practices Every iOS Developer Should Know in 2025"* – on safe optional usage [5] [6].
- Stack Overflow – discussion on polling loops and proper alternatives (use of Timer vs. while-loop) [2].
- Reddit r/swift – commentary on why polling loops with `await` are inefficient (thread tied up doing nothing) [1].
- Sathsara Dharmarathna, *Understanding Multithreading in Swift* – best practices for threading (never block main thread, etc.) [3].
- DeepSource Swift Rules – explanation on why `try!` (force try) is dangerous and should be avoided [7].
- M. Shahzad Qamar, *Swift iOS Tip: Use weak self in Closures to Prevent Retain Cycles* – demonstrating [weak self] to avoid memory leaks [12].
- Apix, *5 SwiftUI Best Practices Every iOS Developer Should Master* – SwiftUI optimization tips (avoiding unnecessary re-renders, using `@StateObject`, lazy stacks, avoiding AnyView) [17] [20].
- Ricardo Santos, *Using SwiftLint to reduce code smell on Xcode* – using SwiftLint rules to catch code smells like prints and commented code [23] [24].
- Russell Stephens, *Swift Smells* – introduction to the concept of code smells and why eliminating them matters [26].

---

[1] Is this the right way to write a Task that loop indefinitely? : r/swift
https://www.reddit.com/r/swift/comments/wqx16l/is_this_the_right_way_to_write_a_task_that_loop/

[2] grand central dispatch - Proper way to do polling in swift? - Stack Overflow
https://stackoverflow.com/questions/44368019/proper-way-to-do-polling-in-swift

[3] Understanding Multithreading Programming in Swift | by Sathsara Dharmarathna | Medium
https://medium.com/@sathsaramadurangad.d/understanding-multithreading-programming-in-swift-fea5c9ffc444

[4] [5] [6] Swift Best Practices Every iOS Developer Should Know in 2025 | by Halil Özel | Medium
https://halilozel1903.medium.com/swift-best-practices-every-ios-developer-should-know-in-2025-b35b69d8dc23

7 8 9 `try!` statements should be avoided (SW-W1008) • Swift
https://deepsource.com/directory/swift/issues/SW-W1008

10 11 12 13 Swift iOS Tip: Use weak Self in Closures to Prevent Retain Cycles | by M Shahzad Qamar | Medium
https://medium.com/@qmshahzad/swift-ios-tip-use-weak-self-in-closures-to-prevent-retain-cycles-bbf719218a08

14 15 16 17 18 19 20 21 22 5 SwiftUI Best Practices Every iOS Developer Should Master | by Apix | Medium
https://medium.com/@vrxrszsb/5-swiftui-best-practices-every-ios-developer-should-master-in-2024-4af4096a0856

23 24 25 Using SwiftLint to reduce code smell on Xcode. | by Ricardo Santos | Medium
https://ricardojpsantos.medium.com/using-swiftlint-to-decrease-code-smell-on-xcode-e1dd49258f22

26 Swift Smells. Identifying Areas for Refactoring in... | by Russell Stephens | Compass True North | Medium
https://medium.com/compass-true-north/swift-smells-17246905d1d9