Specification-Based Banner Control (Swift Example)

This document outlines a modern Swift approach for controlling banner display behavior in an iOS app using the **Specification Pattern** with **Swift Macros** and **Property Wrappers**.

🎯 Goal

Control the display of a **promo banner** using the following rules:

- Show **after 10 seconds** since app launch.
- Show **no more than once per week**.
- Show **no more than 3 times in total**.

✅ Specification Components

We define each rule as a separate **Specification** struct implementing a shared protocol.

```swift
protocol Specification<T> {
    func isSatisfiedBy(_ candidate: T) -> Bool
}
```

Context Object

```swift
struct PromoContext {
    let timeSinceLaunch: TimeInterval
    let lastShownAt: Date?
    let totalShowCount: Int
    let now: Date
}
```

Specification Implementations

```swift
struct DelaySinceLaunchSpec: Specification<PromoContext> {
    let seconds: TimeInterval
    func isSatisfiedBy(_ ctx: PromoContext) -> Bool {
        ctx.timeSinceLaunch >= seconds
    }
}

struct MaxDisplayCountSpec: Specification<PromoContext> {
    let limit: Int
    func isSatisfiedBy(_ ctx: PromoContext) -> Bool {
        ctx.totalShowCount < limit
    }
}

struct CooldownSpec: Specification<PromoContext> {
    let minInterval: TimeInterval
    func isSatisfiedBy(_ ctx: PromoContext) -> Bool {
        guard let last = ctx.lastShownAt else { return true }
        return ctx.now.timeIntervalSince(last) >= minInterval
    }
}
```

⚙️ Swift Macro: `@specs(...)`

Attached macro that **auto-generates** a composite specification:

```swift
@specs(
  DelaySinceLaunchSpec(seconds: 10),
  MaxDisplayCountSpec(limit: 3),
  CooldownSpec(minInterval: .week)
)
struct PromoBannerSpec: Specification<PromoContext> { }
```

Expanded Code (Generated)

```swift
struct PromoBannerSpec: Specification<PromoContext> {
    private let composite: AnySpecification<PromoContext>

    init() {
        composite = AnySpecification(
            DelaySinceLaunchSpec(seconds: 10)
                .and(MaxDisplayCountSpec(limit: 3))
                .and(CooldownSpec(minInterval: 604800)) // 1 week
        )
    }

    func isSatisfiedBy(_ ctx: PromoContext) -> Bool {
        composite.isSatisfiedBy(ctx)
    }
}
```

📦 Property Wrapper: `@Satisfies`

```swift
@propertyWrapper
struct Satisfies<Spec: Specification<Context>, Context> {
    private let context: Context
    private let spec: Spec

    var wrappedValue: Bool {
        spec.isSatisfiedBy(context)
    }

    init(context: Context, using: Spec.Type) {
        self.context = context
        self.spec = Spec()
    }
}
```

📱 Usage in App Code

```swift
let ctx = PromoContext(
    timeSinceLaunch: 12,
    lastShownAt: Date().addingTimeInterval(-900000),
    totalShowCount: 1,
    now: Date()
)

@Satisfies(context: ctx, using: PromoBannerSpec.self)
var shouldShowPromo: Bool

if shouldShowPromo {
```

```
    showPromoBanner()
}
```

🧠 Benefits

- Declarative and reusable logic
- Modular specifications
- Easy to test and maintain
- Clean Swift syntax with Macros and Wrappers

🔧 Next Steps

- Define macro `@specs(...)` via `swift-syntax` plugin
- Expand logic with `AND` , `OR` , `NOT` via operators
- Package as `SpecificationKit` (SPM-ready)

Using a Context Provider with @Satisfies Property Wrapper

This document explains how to eliminate manual context construction for the `@Satisfies` property wrapper by introducing a `PromoContextProvider` .

🎯 Problem

Manually injecting context like this:

```
@Satisfies(context: ctx, using: PromoBannerSpec.self)
var shouldShowPromo: Bool
```

...quickly becomes repetitive and hard to manage across a large codebase.

✅ Solution: Context Provider

Encapsulate context construction logic in a reusable provider object.

👷 1. Define Context Provider Protocol

```
protocol PromoContextProviding {
    func currentContext() -> PromoContext
}
```

Default Implementation

```
struct DefaultPromoContextProvider: PromoContextProviding {
    func currentContext() -> PromoContext {
        PromoContext(
```

```
            timeSinceLaunch: AppState.shared.timeSinceLaunch,
            lastShownAt: AppStorageProvider.lastPromoDate,
            totalShowCount: AppStorageProvider.promoShowCount,
            now: Date()
        )
    }
}
```

🧰 2. Extend Property Wrapper

```
@propertyWrapper
struct Satisfies<Spec: Specification<Context>, Context> {
    private let context: Context
    private let spec: Spec

    var wrappedValue: Bool {
        spec.isSatisfiedBy(context)
    }

    init(provider: some ContextProviding, using: Spec.Type) where Spec.Context == Context {
        self.context = provider.currentContext()
        self.spec = Spec()
    }
}
```

(You may define a `ContextProviding` protocol with `associatedtype Context` to make it generic.)

📱 3. Usage Example

```
@Satisfies(provider: DefaultPromoContextProvider(), using: PromoBannerSpec.self)
var shouldShowPromo: Bool

if shouldShowPromo {
    showBanner()
}
```

🔄 Optional Extension: Global Provider

To make usage even shorter:

```
extension PromoContextProviding {
    static var shared: some PromoContextProviding {
        DefaultPromoContextProvider()
    }
}
```

Then you can write:

```
@Satisfies(provider: .shared, using: PromoBannerSpec.self)
var shouldShowPromo: Bool
```

💡 Benefits

- No manual context assembly in views or services

- Central place to adjust logic if data structure changes
- Easily mockable in unit tests

🧪 Bonus: Mock Provider for Testing

```
struct MockPromoContextProvider: PromoContextProviding {
    let context: PromoContext
    func currentContext() -> PromoContext { context }
}
```

Use in tests:

```
@Satisfies(provider: MockPromoContextProvider(context: testContext), using: PromoBannerSpec.self)
var result: Bool
```

✅ Summary

Introducing a `PromoContextProvider` helps decouple logic from UI and allows reusable, testable, and concise `@Satisfies` declarations throughout your app.

AutoContextSpecification Pattern in Swift

This document outlines how to eliminate manual context boilerplate in `@Satisfies` property wrappers by encapsulating context logic inside specification types using a protocol called `AutoContextSpecification`.

🎯 Goal

Instead of writing:

```
@Satisfies(provider: DefaultPromoContextProvider(), using: PromoBannerSpec.self)
var shouldShowPromo: Bool
```

We want to simplify to:

```
@Satisfies(using: PromoBannerSpec.self)
var shouldShowPromo: Bool
```

The context provider is internal to the specification.

🧩 Step 1: Define `ContextProviding`

```
protocol ContextProviding {
    associatedtype Context
    func currentContext() -> Context
}
```

This makes context generation reusable and testable.

🧩 Step 2: Define `AutoContextSpecification`

```
protocol Specification {
    associatedtype Context
    func isSatisfiedBy(_ context: Context) -> Bool
}


protocol AutoContextSpecification: Specification {
    associatedtype Provider: ContextProviding where Provider.Context == Context
    static var contextProvider: Provider { get }
    init()
}
```

This allows each spec to declare its own context logic.

🧩 Step 3: Update `@Satisfies` Property Wrapper

```
@propertyWrapper
struct Satisfies<Spec: AutoContextSpecification> {
    private let value: Bool
    var wrappedValue: Bool { value }

    init(using specType: Spec.Type) {
        let context = specType.contextProvider.currentContext()
        let spec = specType.init()
        self.value = spec.isSatisfiedBy(context)
    }
}
```

> ⚠️ The `Spec` type must be `init()` -able.

🏗️ Step 4: Implement a Spec with a Provider

```
struct PromoContext {
    let timeSinceLaunch: TimeInterval
    let lastShownAt: Date?
    let totalShowCount: Int
    let now: Date
}


struct DefaultPromoContextProvider: ContextProviding {
    func currentContext() -> PromoContext {
        PromoContext(
            timeSinceLaunch: AppState.shared.timeSinceLaunch,
            lastShownAt: AppStorageProvider.lastPromoDate,
            totalShowCount: AppStorageProvider.promoShowCount,
            now: Date()
        )
    }
}


struct PromoBannerSpec: AutoContextSpecification {
    static let contextProvider = DefaultPromoContextProvider()

    private let composite: AnySpecification<PromoContext>

    init() {
        composite = AnySpecification(
            DelaySinceLaunchSpec(seconds: 10)
                .and(MaxDisplayCountSpec(limit: 3))
```

```
                .and(CooldownSpec(minInterval: 604800))
        }
    }

    func isSatisfiedBy(_ ctx: PromoContext) -> Bool {
        composite.isSatisfiedBy(ctx)
    }
}
```

✅ Final Usage

```
@Satisfies(using: PromoBannerSpec.self)
var shouldShowPromo: Bool
```

🧪 Bonus: Swapping Providers in Tests

Make `contextProvider` a mutable static var:

```
static var contextProvider: PromoContextProviding = DefaultPromoContextProvider()
```

Then in tests:

```
PromoBannerSpec.contextProvider = MockPromoProvider()
```

✅ Summary

Feature	Benefit
Internalized context	Removes boilerplate from caller
Generic provider system	Reusable and testable
Declarative usage	`@Satisfies(using: ...)` reads cleanly
Test override ready	Easily mock context during unit tests

This pattern helps build expressive, reusable, and testable conditional logic for feature flags, access control, banners, etc.