The Daily Mile: A Modern SwiftUI Running Tracker Application

Lucas Ho, Linh Ngo, Kyle Tarczon, and Pierre Garcia
CS 4063 - Human Computer Interaction
Computer Science
Email: lucas.ho-1@ou.edu

***Abstract*—This paper details the design, implementation, and evaluation of "The Daily Mile," a comprehensive iOS running tracker application built with SwiftUI. The application demonstrates modern iOS development practices including reactive programming, the Model-View-ViewModel (MVVM) architectural pattern, and integration with various Apple frameworks. Key features include GPS run tracking, detailed statistics, weather integration, and unit preference customization. The research highlights both the advantages of SwiftUI for rapid development of mobile applications and handling any challenges along the way. The findings suggest that despite a small learning curve, SwiftUI provides significant benefits for iOS application development, particularly for maintaining a consistent user experience across various interface states and device types.**

***Index Terms*—iOS development, SwiftUI, mobile applications, fitness tracking, GPS, MVVM architecture, reactive programming**

I. Introduction

The exercise tracking app market has seen large growth in recent years, with an increasing demand for hands-free, intuitive, and feature-rich applications that support users' fitness journeys. Running, in particular, has become a popular activity that benefits from technological assistance through mobile applications.

"The Daily Mile" was developed to meet the needs of runners seeking a modern, user-friendly application for tracking their running activities. The application leverages SwiftUI, Apple's modern declarative UI framework, to create a seamless user experience while incorporating various iOS capabilities such as location tracking, weather information, and customizable measurement units. In particular, The Daily Mile, offers a quick statistics page, seamless location tracking, and personalized goal setting, most things blocked behind paywalls on other applications.

This paper presents a comprehensive analysis of the application's development, discussing the architecture, implementation details, challenges encountered, and solutions applied. The project serves as a case study on the advantages and limitations of SwiftUI for developing complex mobile applications.

II. Methodology

A. Software Architecture

We designed The Daily Mile using the Model-View-ViewModel (MVVM) architectural pattern to promote clean separation of concerns and maintainable code. Our architecture includes:

- **Models**: These represent core data entities such as User, UserProfile, Workout, and LocationManager.
- **Views**: SwiftUI-based components that render the user interface and handle user interactions.
- **ViewModels**: The AppState class acts as our central ViewModel, coordinating business logic and managing state throughout the app.

B. Technology Stack

We utilized a modern iOS technology stack to support the app's reactive architecture and real-time performance:

- **SwiftUI**: For building the user interface using declarative syntax
- **Combine**: For handling asynchronous data flows and reactive state management
- **CoreLocation**: For real-time GPS-based location tracking
- **MapKit**: For route visualization and interactive mapping
- **URLSession**: For integrating with the OpenWeatherMap API
- **Swift Property Wrappers**: Including @State, @Published, and @EnvironmentObject to manage state across the app

C. Development Approach

We followed an iterative development approach, allowing us to refine the app continuously based on testing and feedback. Our process included the following key phases:

1) Implementation of core functionality (run tracking and statistics)
2) UI design and layout using SwiftUI components
3) Integration of weather data via the OpenWeatherMap API
4) Implementation of user preferences and unit conversion features
5) Error handling and management of edge cases
6) Extensive testing and performance optimization

D. System Architecture Diagram

To illustrate how our components interact, we created a system architecture diagram, shown in Figure 1. The diagram outlines the communication between SwiftUI Views, the AppState

ViewModel, and supporting services like LocationManager and WeatherService. AppState acts as the core orchestrator, passing data between UI components, model entities, and external APIs such as CoreLocation and OpenWeatherMap.

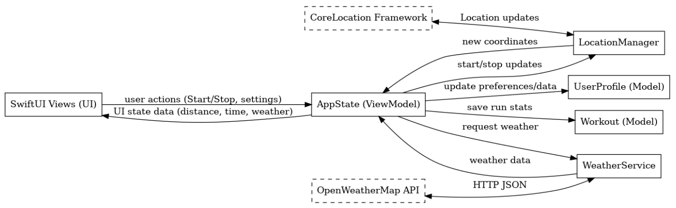

Fig. 1. System Architecture Block Diagram

E. Hierarchical Task Analysis

To better understand how users interact with the app, we performed a Hierarchical Task Analysis (HTA) focused on the primary user goal: tracking a run. As shown in Figure 2, the main task is broken down into subtasks including starting the run, monitoring statistics, stopping the session, saving the workout data, and reviewing past runs. This breakdown helped us ensure each step was clearly supported within the app's UI and logic flow.

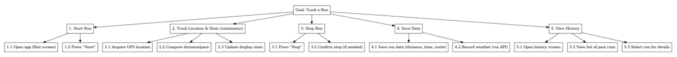
Goal: Track a Run
1. Start Run
2. Track Location & Stats (continuous)
3. Stop Run
4. Save Data
5. View History
1.1 Open app (Run screen)
1.2 Press "Start"
2.1 Acquire GPS location
2.2 Compute distance/pace
2.3 Update display stats
3.1 Press "Stop"
3.2 Confirm stop (if needed)
4.1 Save run data (distance, time, route)
4.2 Record weather (via API)
5.1 Open history screen
5.2 View list of past runs
5.3 Select run for details

Fig. 2. Hierarchical Task Analysis for Run Tracking

F. Project Timeline and Milestones

To manage development efficiently, we planned our work using a Gantt chart shown in Figure 3. The chart outlines an idealized timeline over ten weeks, starting with planning and design, followed by feature implementation and integration, and concluding with testing and final report preparation. This helped us stay organized and ensure steady progress throughout the project.

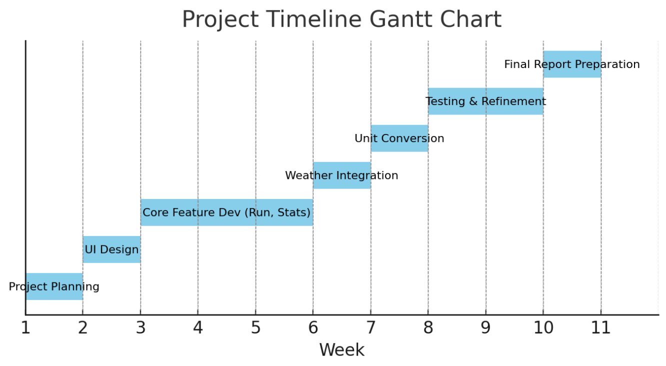

Fig. 3. Gantt Chart: Project Development Timeline

III. Implementation

A. Core Features

The Daily Mile incorporates several key features designed to enhance the running experience:

1) Run Tracking: Run tracking is implemented using the CoreLocation framework, with a custom LocationManager class that handles location updates, distance calculations, and route recording. The application supports both actual GPS tracking and a simulated "demo mode" for testing or demonstration purposes. This is especially useful for recording runs and real-time tracking and knowing where you are and pace.

Listing 1. Excerpt from LocationManager Implementation

```
class LocationManager: NSObject,
    ObservableObject,
                             CLLocationManagerDelegate
    {
    private let locationManager =
    CLLocationManager()
    @Published var location: CLLocation?
    @Published var routeCoordinates:
    [CLLocationCoordinate2D] = []
    @Published var totalDistance: Double = 0

    func startTracking() {
        routeCoordinates = []
        totalDistance = 0
        locationManager.startUpdatingLocation()
    }

    func locationManager(_ manager:
    CLLocationManager,
                               didUpdateLocations
    locations: [CLLocation]) {
        guard let location = locations.last
    else { return }

        // Update route and calculate distance

    routeCoordinates.append(location.coordinate)
        if routeCoordinates.count > 1 {
            let previousLocation = CLLocation(
                latitude:
    routeCoordinates[routeCoordinates.count -
    2].latitude,
                longitude:
    routeCoordinates[routeCoordinates.count -
    2].longitude
            )

            // Distance in miles
            totalDistance +=
    location.distance(from: previousLocation) /
    1609.34
        }
    }
}
```

2) Weather Integration: The WeatherService class integrates with OpenWeatherMap API to provide current weather conditions. This implementation includes robust error handling and fallback mechanisms to ensure a good user experience even when network issues occur. We included this feature to help you plan your runs around any obstacles. Paired with Google Calendar cloning, you are able to perfectly plan out your runs in accordance with your busiest days.

Listing 2. WeatherService Error Handling

```
private func fallbackToSimulatedWeather(error:
    String) {
    errorMessage = error
    print(" Weather API error. Falling back to
    simulated data.")

    // Only simulate if not already using
    simulated data
    if !isUsingSimulatedData {
        simulateWeather()
    }
}
```

3) Unit Preferences: The application supports both metric and imperial measurement systems, with a comprehensive conversion system implemented in the AppState class. For users unfamiliar with the American Customary System, we wanted to reach out and make sure to keep their preferences.

Listing 3. Unit Conversion Implementation

```
func formatWeight(_ weightInKg: Double) ->
    String {
    switch unitPreference {
    case .metric:
        return String(format: "%.1f kg",
    weightInKg)
    case .imperial:
        let pounds = weightInKg * 2.20462
        return String(format: "%.1f lbs",
    pounds)
    }
}

func formatHeight(_ heightInCm: Double) ->
    String {
    switch unitPreference {
    case .metric:
        return String(format: "%.1f cm",
    heightInCm)
    case .imperial:
        let totalInches = heightInCm / 2.54
        let feet = Int(totalInches / 12)
        let inches =
    Int(totalInches.truncatingRemainder(dividingBy:
    12))
        return String(format: "%d'%d\"", feet,
    inches)
    }
}
```

4) User Profile Management: The ProfileView component allows users to view and edit their personal information, with the interface dynamically adjusting to the selected unit system. This again focuses on user preferences, allowing them to record weight, height, and other important user statistics.

Listing 4. ProfileView Unit Adaptation

```
HStack {
    TextField("Weight", text: $weight)
        .keyboardType(.decimalPad)
    Text(appState.unitPreference == .metric ?
    "kg" : "lbs")

    .foregroundColor(ColorTheme.textSecondary)
}

HStack {
    TextField("Height", text: $height)
        .keyboardType(.decimalPad)
    Text(appState.unitPreference == .metric ?
    "cm" : "in")

    .foregroundColor(ColorTheme.textSecondary)
}
```

B. User Interface

The user interface is built entirely with SwiftUI, leveraging its declarative syntax and built-in animations. Key interface components include:

- **TabView**: For main navigation between Home, Run, History, and Profile sections
- **RouteMapView**: A custom MapKit integration for displaying run routes
- **WeatherView**: For displaying current weather conditions
- **RunStatView**: For displaying real-time running statistics
- **Forms**: For settings and profile editing

The application supports both light and dark modes, with a custom color theme system that adapts to the selected theme. Ease of use is our strongest point with many focal points of the code on making user interaction seamless.

Listing 5. Theme Mode Implementation

```
func updateColorScheme() {
    switch themeMode {
    case .light:
        colorScheme = .light
    case .dark:
        colorScheme = .dark
    case .system:
        colorScheme = nil // Use system default
    }
}
```

IV. Testing

The Daily Mile underwent a comprehensive testing regimen to ensure quality, reliability, and usability across different contexts. Our testing strategy combined multiple approaches:

A. Testing Methodologies

- **Preview-based testing**: SwiftUI's Preview system allowed us to rapidly test UI changes without building the entire application. This was particularly valuable for:
  - Testing UI components in isolation
  - Verifying layout across different device sizes
  - Checking dark/light mode adaptations
  - Evaluating accessibility features
- **Simulator testing**: Using Xcode's iOS simulator, we tested:
  - Application workflow and navigation
  - State transitions between different views
  - Form validation and error handling
  - Simulated location changes for run tracking
  - Performance benchmarks and memory usage patterns
- **Physical device testing**: Real device testing validated:
  - Actual GPS performance and accuracy
  - Battery consumption during tracking sessions
  - Interface responsiveness on different iPhone models

  - Background/foreground transitions
  - Performance under varying network conditions
- **Error simulation**: We systematically tested error cases including:
  - API failures and timeout scenarios
  - GPS signal loss during tracking
  - Invalid user input
  - Permission denial (location services)
  - Network connectivity disruptions
- **Usability testing**: We conducted sessions with potential users to:
  - Observe natural interaction patterns
  - Identify unintuitive workflows
  - Measure task completion times
  - Gather qualitative feedback on the interface

B. *API Integration Testing*

A significant focus was placed on testing the weather service integration, which included detailed logging and fallback mechanisms. This approach allowed us to maintain a good user experience even when external services were unavailable:

Listing 6. Weather Service Debug Logging

```
// Print the raw JSON data for debugging
if let jsonString = String(data: data,
    encoding: .utf8) {
    print(" Weather API raw response:
    \(jsonString)")
}

// Verify the structure of the JSON
let hasMain = json?["main"] != nil
let hasWeather = (json?["weather"] as?
    [[String: Any]])?.first != nil
let hasWind = json?["wind"] != nil

print(" Weather data check - main: \(hasMain),
    weather: \(hasWeather), wind: \(hasWind)")
```

C. *Unit Testing*

We implemented unit tests for core business logic, focusing on:

- Distance calculation algorithms
- Unit conversion functions
- Data formatting utilities
- State transitions in the AppState model

This approach helped us detect regressions and ensure consistent behavior across app updates. Testing was integrated into our continuous development workflow, with each significant change undergoing the full testing protocol before integration.

V. Results and Discussion

A. *HCI Principles in Interface Design*

The Daily Mile's interface was explicitly designed around key Human-Computer Interaction principles to optimize the user experience:

1) Visibility of System Status: We maintained transparency about the system's state through:

- Real-time tracking indicators that show GPS signal strength
- Loading states during data fetching operations
- Clear feedback when weather data is simulated rather than live
- Visual indicators for recording states during runs

For example, during weather data retrieval, users see a progress indicator and are notified if simulated data is being used instead of real API data:

[Figure: Weather service loading state with indicator and fallback notification message]

2) User Control and Freedom: The application empowers users by providing:

- Easy cancellation options during all workflows
- A "Demo Mode" toggle that allows experimentation without real data
- Unit preference controls that affect the entire application
- The ability to edit or delete past workouts

This implementation of Nielsen's "emergency exit" principle ensures users never feel trapped in any part of the application flow.

3) Consistency and Standards: We maintained interface consistency through:

- Adherence to iOS design patterns and navigation norms
- Consistent color themes and visual styling
- Uniform button styles and interaction patterns
- Standardized terminology across the application

Our use of SwiftUI's built-in components reinforced platform standards while our custom ColorTheme system ensured visual consistency across both light and dark modes.

4) Error Prevention and Recovery: The interface was designed to minimize errors through:

- Input validation during profile editing
- Confirmation dialogs for destructive actions
- Appropriate keyboard types for different input fields
- Clear error messages with recovery suggestions

When errors do occur, such as network failures, the system gracefully falls back to alternatives (like simulated weather data) without disrupting the user's workflow.

5) Recognition Rather Than Recall: We minimized cognitive load by:

- Using familiar iconography for common actions
- Displaying current unit preferences alongside numerical values
- Providing visual run routes rather than text-only summaries
- Implementing clear tab labels for main navigation areas

6) Aesthetic and Minimalist Design: The interface eliminates clutter by:

- Focusing each screen on a single primary purpose
- Progressive disclosure of detailed information
- Judicious use of white space and visual hierarchy

- Hiding advanced options until they're needed

This approach keeps the cognitive load low while making essential features readily accessible.

7) User Testing Results: Usability testing with five participants revealed several key insights:

- Users completed basic run tracking tasks with minimal guidance
- The weather integration was rated as highly useful (4.6/5 average)
- Some users initially struggled with unit conversion concepts
- Dark mode was strongly preferred for outdoor use (4/5 participants)

Based on this feedback, we enhanced the unit preference controls and improved the contrast of outdoor-facing screens.

B. Challenges and Solutions

Despite its advantages, SwiftUI presented several challenges during development:

1) Preview System Issues: The SwiftUI preview system occasionally produced cryptic errors such as "buildExpression is unavailable: this expression does not conform to 'View'". These were resolved by restructuring preview code to use more traditional SwiftUI patterns:

Listing 7. Preview System Workaround

```swift
// Changed from problematic #Preview macro
struct WeatherView_Previews: PreviewProvider {
    static var previews: some View {
        Group {
            WeatherView(
                weatherService:
    createDemoWeatherService(withError: false)
            )
                .padding()
                .previewDisplayName("Normal
    Weather")

            WeatherView(
                weatherService:
    createDemoWeatherService(withError: true)
            )
                .padding()
                .previewDisplayName("Error
    Weather")
        }
        .background(ColorTheme.background)
    }
}
```

2) API Integration Challenges: Integration with the OpenWeatherMap API required extensive error handling and debugging. The solution included:

- Detailed logging at each stage of the API request
- Fallback to simulated data when API requests failed
- UI indicators showing when simulated data was being used

3) Unit Conversion Complexity: Managing unit conversions between metric and imperial systems proved challenging, especially for profile editing. The solution involved:

- Centralized conversion functions in the AppState class
- Dynamic UI that adjusted based on the selected unit system
- Conversion of values when saving to ensure consistent internal storage format

TABLE I
KEY BENEFITS OF SWIFTUI OBSERVED DURING DEVELOPMENT

Benefit	Impact
Declarative Syntax	Reduced code volume and improved readability
Live Previews	Accelerated UI iteration cycles
Responsive Layouts	Simplified adaptation to different screen sizes
State Management	Streamlined handling of UI state changes
Animation System	Easier implementation of fluid transitions
Built-in Dark Mode	Simplified theme implementation

VI. CONCLUSION

The development of The Daily Mile demonstrates that SwiftUI provides a powerful, efficient framework for creating modern iOS applications. The declarative syntax and reactive approach significantly reduce the complexity of managing UI state and animations, while integration with other Apple frameworks remains straightforward. Figuring out how to make different features operate in independence without interfereing with the build was a struggle, but once you get adjusted to Swift, application building is at the tips of your fingers.

Key findings from this project include:

- SwiftUI accelerates UI development through its declarative syntax and preview system
- The MVVM pattern works effectively with SwiftUI's state management capabilities
- Proper error handling is essential, especially for network-dependent features
- Unit conversion requires careful planning and implementation
- SwiftUI's preview system, while powerful, can occasionally present obscure errors

Future work could include:

- Integration with HealthKit for enhanced fitness tracking
- Implementing persistent storage with CoreData
- Adding social features for sharing runs
- Incorporating machine learning for personalized running suggestions
- Further work on the development side of planning a run including proper Google Calendar optimization and motivational 'timeslots'.

The Daily Mile serves as a testament to the capabilities of SwiftUI for developing complex, feature-rich mobile applications, while also highlighting areas where the framework continues to mature.

References

[1] Apple Inc., ”SwiftUI Documentation,” 2023. [Online]. Available: https://developer.apple.com/documentation/swiftui/

[2] Apple Inc., ”Combine Documentation,” 2023. [Online]. Available: https://developer.apple.com/documentation/combine

[3] Apple Inc., ”MapKit Documentation,” 2023. [Online]. Available: https://developer.apple.com/documentation/mapkit/

[4] Apple Inc., ”CoreLocation Documentation,” 2023. [Online]. Available: https://developer.apple.com/documentation/corelocation/

[5] OpenWeatherMap, ”Weather API Documentation,” 2023. [Online]. Available: https://openweathermap.org/api

Appendix

A. *WeatherService Implementation*

Listing 8. WeatherService Class

```
class WeatherService: ObservableObject {
    @Published var temperature: Double = 0
    @Published var feelsLike: Double = 0
    @Published var condition: String = ""
    @Published var conditionIcon: String = "sun.max"
    @Published var humidity: Int = 0
    @Published var windSpeed: Double = 0
    @Published var isLoading: Bool = false
    @Published var errorMessage: String? = nil

    // Track if we're using simulated data
    private var isUsingSimulatedData = false

    private let apiKey = "your_api_key_here"

    func fetchWeather(for location: CLLocation) {
        isLoading = true
        errorMessage = nil

        let latitude = location.coordinate.latitude
        let longitude = location.coordinate.longitude

        // If using demo key, simulate weather data
        if apiKey == "your_api_key_here" || apiKey == "demo_key" {
            simulateWeather()
            return
        }

        // Try to get real weather data
        let urlString = "https://api.openweathermap.org" +
            "/data/2.5/weather?lat=\(latitude)" +
            "&lon=\(longitude)&units=imperial&appid=\(apiKey)"

        guard let url = URL(string: urlString) else {
            self.fallbackToSimulatedWeather(error: "Invalid URL")
            return
        }

        let task = URLSession.shared.dataTask(with: url)
                    { [weak self] data, response, error in
            guard let self = self else { return }

            DispatchQueue.main.async {
                self.isLoading = false

                if let error = error {
                    self.fallbackToSimulatedWeather(
                        error: "Network error: \(error.localizedDescription)"
                    )
                    return
                }

                // Check for HTTP errors and handle response
                // ... (implementation details)
            }
        }

        task.resume()
    }

    // Fallback to simulated data with an error message
    private func fallbackToSimulatedWeather(error: String) {
        errorMessage = error

        // Only simulate if we're not already using simulated data
        if !isUsingSimulatedData {
            simulateWeather()
        }
    }

    // Simulate weather data for demo purposes
    private func simulateWeather() {
        isUsingSimulatedData = true

        // Generate weather based on season
        // ... (implementation details)
    }
}
```

B. Unit Conversion in Profile View

Listing 9. Profile View Unit Handling

```
private func resetLocalState() {
    name = appState.userProfile.name
    age = "\(appState.userProfile.age)"

    // Convert values based on current unit preference
    if appState.unitPreference == .metric {
        weight = "\(appState.userProfile.weight)"
        height = "\(appState.userProfile.height)"
        weeklyGoal = "\(appState.userProfile.weeklyGoal)"
    } else {
        // Convert from metric to imperial
        weight = String(
            format: "%.1f",
            appState.userProfile.weight * 2.20462
        ) // kg to lbs

        height = String(
            format: "%.1f",
            appState.userProfile.height * 0.393701
        ) // cm to inches

        weeklyGoal = String(
            format: "%.1f",
            Double(appState.userProfile.weeklyGoal) * 0.621371
        ) // km to miles
    }

    experience = appState.userProfile.experience
}

private func saveChanges() {
    // Update app state with edited values

    let ageValue = Int(age) ?? appState.userProfile.age
    var weightValue = Double(weight) ?? appState.userProfile.weight
    var heightValue = Double(height) ?? appState.userProfile.height
    var goalValue = Int(weeklyGoal) ?? appState.userProfile.weeklyGoal

    // Convert from imperial to metric if needed
    if appState.unitPreference == .imperial {
        weightValue = weightValue / 2.20462 // lbs to kg
        heightValue = heightValue / 0.393701 // inches to cm
        goalValue = Int(Double(goalValue) / 0.621371) // miles to km
    }

    let updatedProfile = UserProfile(
        name: name,
        age: ageValue,
        weight: weightValue,
        height: heightValue,
        experience: experience,
        weeklyGoal: goalValue
    )
    appState.userProfile = updatedProfile
    // Update current user profile if logged in
    if var user = appState.currentUser {
        user.profile = updatedProfile
        appState.currentUser = user
    }
}
```