# Chapter 4: System Design

## 4.1 Design Pseudocode or Algorithm for Method or Operation

This section details the key algorithms and methods used in the Pro Mood Tracker application. The pseudocode presented here illustrates the core logic behind crucial operations within the system.

### 4.1.1 Mood Entry Algorithm

The mood entry process is one of the most frequently used features in the application. The following pseudocode describes the algorithm for creating and saving a new mood entry:

FUNCTION createMoodEntry(mood, timeOfDay, date, notes, activities, weatherTracking):  
 // Validate required inputs  
 IF mood IS NULL OR NOT IN ["Very Bad", "Bad", "Okay", "Good", "Very Good"] THEN  
 RETURN Error("Invalid mood selection")  
 END IF  
   
 IF timeOfDay IS NULL OR NOT IN ["morning", "afternoon", "evening", "night", "full-day"] THEN  
 RETURN Error("Invalid time of day")  
 END IF  
   
 // Create the entry object with required fields  
 entry = {  
 id: generateUUID(),  
 date: formatISO(date || currentDate()),  
 timeOfDay: timeOfDay,  
 mood: mood,  
 notes: notes || "",  
 activities: activities || [],  
 createdAt: formatISO(currentDateTime())  
 }  
   
 // Add weather data if enabled  
 IF weatherTracking IS TRUE THEN  
 TRY  
 weatherData = getWeatherData(userLocation)  
 entry.weather = {  
 temperature: weatherData.temperature,  
 condition: weatherData.condition,  
 humidity: weatherData.humidity  
 }  
 CATCH error  
 // Weather data is optional, continue without it  
 LogWarning("Could not retrieve weather data: " + error.message)  
 END TRY  
 END IF  
   
 // Save to storage  
 TRY  
 existingEntries = getFromStorage("proMoodTracker:moodEntries") || []  
   
 // Check for duplicate entries on same date and timeOfDay  
 IF hasDuplicateEntry(existingEntries, date, timeOfDay) THEN  
 RETURN Confirmation("Entry exists for this time period. Replace?")  
 IF confirmed THEN  
 existingEntries = removeExistingEntry(existingEntries, date, timeOfDay)  
 ELSE  
 RETURN Error("Canceled by user")  
 END IF  
 END IF  
   
 // Add new entry and save  
 updatedEntries = existingEntries.concat(entry)  
 saveToStorage("proMoodTracker:moodEntries", updatedEntries)  
   
 // Update streaks  
 updateStreaks(date)  
   
 // Check for achievements  
 checkAchievements(updatedEntries)  
   
 RETURN Success(entry)  
 CATCH error  
 RETURN Error("Failed to save entry: " + error.message)  
 END TRY  
END FUNCTION

### 4.1.2 Streak Calculation Algorithm

Streaks are an important part of the application’s engagement strategy. The following pseudocode illustrates how the application calculates and updates user streaks:

FUNCTION updateStreaks(entryDate):  
 // Get current streak information  
 streakData = getFromStorage("proMoodTracker:streaks") || {  
 currentDaily: 0,  
 longestDaily: 0,  
 startDate: null,  
 lastEntryDate: null,  
 history: []  
 }  
   
 // If this is the first entry ever  
 IF streakData.lastEntryDate IS NULL THEN  
 streakData.currentDaily = 1  
 streakData.longestDaily = 1  
 streakData.startDate = formatISO(entryDate)  
 streakData.lastEntryDate = formatISO(entryDate)  
 saveToStorage("proMoodTracker:streaks", streakData)  
 RETURN streakData  
 END IF  
   
 lastEntryDate = parseISO(streakData.lastEntryDate)  
 currentDate = parseISO(entryDate)  
   
 // Calculate days between last entry and current entry  
 dayDifference = differenceInDays(currentDate, lastEntryDate)  
   
 // If entry is for today (duplicate entry), streak doesn't change  
 IF dayDifference = 0 THEN  
 RETURN streakData  
 END IF  
   
 // If entry is for tomorrow (continuing streak)  
 IF dayDifference = 1 THEN  
 streakData.currentDaily += 1  
   
 // Update longest streak if current is now longer  
 IF streakData.currentDaily > streakData.longestDaily THEN  
 streakData.longestDaily = streakData.currentDaily  
 END IF  
 // If entry is for a day more than 1 day in the future (missed days)  
 ELSE IF dayDifference > 1 THEN  
 // Record completed streak in history if it was significant  
 IF streakData.currentDaily >= 3 THEN  
 completedStreak = {  
 type: "daily",  
 count: streakData.currentDaily,  
 startDate: streakData.startDate,  
 endDate: streakData.lastEntryDate  
 }  
 streakData.history.push(completedStreak)  
   
 // Limit history size  
 IF streakData.history.length > 10 THEN  
 streakData.history = streakData.history.slice(-10)  
 END IF  
 END IF  
   
 // Reset streak  
 streakData.currentDaily = 1  
 streakData.startDate = formatISO(entryDate)  
 END IF  
   
 // Update last entry date  
 streakData.lastEntryDate = formatISO(entryDate)  
   
 // Save updated streak data  
 saveToStorage("proMoodTracker:streaks", streakData)  
   
 // Check if streak milestone achieved  
 checkStreakMilestones(streakData.currentDaily)  
   
 RETURN streakData  
END FUNCTION

### 4.1.3 Mood Trend Analysis Algorithm

The mood trend analysis is a key feature providing insights to users. The following pseudocode shows how the application analyzes mood trends:

FUNCTION analyzeMoodTrends(timeRange):  
 // Get all mood entries  
 allEntries = getFromStorage("proMoodTracker:moodEntries") || []  
   
 // Filter entries by time range  
 IF timeRange IS NOT NULL THEN  
 startDate = timeRange.startDate  
 endDate = timeRange.endDate || currentDate()  
 filteredEntries = filterEntriesByDateRange(allEntries, startDate, endDate)  
 ELSE  
 // Default to last 30 days if no range specified  
 endDate = currentDate()  
 startDate = subtractDays(endDate, 30)  
 filteredEntries = filterEntriesByDateRange(allEntries, startDate, endDate)  
 END IF  
   
 // If not enough data for meaningful analysis  
 IF filteredEntries.length < 5 THEN  
 RETURN {  
 sufficient: false,  
 message: "Not enough data for analysis. Log more entries for insights."  
 }  
 END IF  
   
 // Convert mood strings to numerical values for calculation  
 numericEntries = convertMoodsToNumeric(filteredEntries)  
   
 // Calculate average mood by day of week  
 averageMoodByDay = calculateAverageByProperty(numericEntries, getDayOfWeek)  
   
 // Calculate average mood by time of day  
 averageMoodByTimeOfDay = calculateAverageByProperty(numericEntries, "timeOfDay")  
   
 // Count occurrences of each mood type  
 moodCountByType = countOccurrencesByProperty(filteredEntries, "mood")  
   
 // Find most frequent mood  
 mostFrequentMood = findKeyWithHighestValue(moodCountByType)  
   
 // Calculate overall trend (improving, declining, stable)  
 moodValues = numericEntries.map(entry => entry.numericValue)  
 moodDates = numericEntries.map(entry => parseISO(entry.date))  
   
 // Simple linear regression for trend  
 regressionResult = calculateLinearRegression(moodDates, moodValues)  
   
 // Determine trend direction  
 IF regressionResult.slope > 0.05 THEN  
 trendDirection = "improving"  
 ELSE IF regressionResult.slope < -0.05 THEN  
 trendDirection = "declining"  
 ELSE  
 trendDirection = "stable"  
 END IF  
   
 // Calculate correlations if enough data points  
 correlations = {}  
 IF filteredEntries.length >= 14 THEN  
 // Weather correlation if available  
 weatherEntries = numericEntries.filter(entry => entry.weather)  
 IF weatherEntries.length >= 10 THEN  
 temperatureCorrelation = calculateCorrelation(  
 weatherEntries.map(e => e.weather.temperature),  
 weatherEntries.map(e => e.numericValue)  
 )  
   
 correlations.weather = {  
 temperature: temperatureCorrelation  
 }  
   
 // Group by weather condition  
 weatherConditions = groupEntriesByProperty(weatherEntries, "weather.condition")  
 weatherCorrelations = {}  
   
 FOREACH condition, entries IN weatherConditions  
 IF entries.length >= 3 THEN  
 avgMood = calculateAverage(entries.map(e => e.numericValue))  
 weatherCorrelations[condition] = avgMood  
 END IF  
 END FOREACH  
   
 correlations.weather.condition = weatherCorrelations  
 END IF  
   
 // Day of week correlation  
 correlations.dayOfWeek = averageMoodByDay  
 END IF  
   
 // Compile and return results  
 RETURN {  
 sufficient: true,  
 timeRange: {  
 startDate: startDate,  
 endDate: endDate  
 },  
 entryCount: filteredEntries.length,  
 averageMoodByDay: averageMoodByDay,  
 averageMoodByTimeOfDay: averageMoodByTimeOfDay,  
 moodCountByType: moodCountByType,  
 mostFrequentMood: mostFrequentMood,  
 moodTrend: trendDirection,  
 trendConfidence: Math.abs(regressionResult.correlation),  
 moodCorrelations: correlations  
 }  
END FUNCTION

### 4.1.4 Achievement Detection Algorithm

The achievement system encourages continued usage. The following pseudocode demonstrates how the application detects when a user has earned an achievement:

FUNCTION checkAchievements(moodEntries):  
 // Get current user badges  
 userBadges = getFromStorage("proMoodTracker:badges") || []  
   
 // Get all available achievements  
 allAchievements = getAvailableAchievements()  
   
 // Get streak data  
 streakData = getFromStorage("proMoodTracker:streaks") || { currentDaily: 0, longestDaily: 0 }  
   
 // Get user data  
 userData = getFromStorage("proMoodTracker:user") || { points: 0 }  
   
 // New badges earned in this check  
 newlyEarnedBadges = []  
   
 FOREACH achievement IN allAchievements  
 // Skip if already earned  
 IF userBadges.some(badge => badge.id === achievement.id) THEN  
 CONTINUE  
 END IF  
   
 // Check if achievement criteria are met  
 achievementEarned = FALSE  
   
 SWITCH achievement.type  
 CASE "entryCount":  
 achievementEarned = moodEntries.length >= achievement.threshold  
 BREAK  
   
 CASE "streakCurrent":  
 achievementEarned = streakData.currentDaily >= achievement.threshold  
 BREAK  
   
 CASE "streakLongest":  
 achievementEarned = streakData.longestDaily >= achievement.threshold  
 BREAK  
   
 CASE "moodVariety":  
 uniqueMoods = countUniqueMoods(moodEntries)  
 achievementEarned = uniqueMoods >= achievement.threshold  
 BREAK  
   
 CASE "journalLength":  
 longJournalEntries = countEntriesWithLongJournals(moodEntries, achievement.threshold)  
 achievementEarned = longJournalEntries >= achievement.minEntries  
 BREAK  
   
 CASE "timeConsistency":  
 consistencyScore = calculateTimeConsistency(moodEntries)  
 achievementEarned = consistencyScore >= achievement.threshold  
 BREAK  
   
 CASE "activityTracking":  
 entriesWithActivities = countEntriesWithActivities(moodEntries)  
 achievementEarned = entriesWithActivities >= achievement.threshold  
 BREAK  
   
 CASE "weatherTracking":  
 entriesWithWeather = countEntriesWithWeather(moodEntries)  
 achievementEarned = entriesWithWeather >= achievement.threshold  
 BREAK  
   
 CASE "customTheme":  
 customThemes = getFromStorage("proMoodTracker:themes") || []  
 achievementEarned = customThemes.length >= achievement.threshold  
 BREAK  
   
 // Add other achievement types as needed  
 END SWITCH  
   
 // If achievement was earned  
 IF achievementEarned THEN  
 // Create badge object  
 newBadge = {  
 id: achievement.id,  
 name: achievement.name,  
 description: achievement.description,  
 imageUrl: achievement.imageUrl,  
 criteria: achievement.criteria,  
 points: achievement.points,  
 tier: achievement.tier,  
 earnedAt: formatISO(currentDateTime())  
 }  
   
 // Add to user's badges  
 userBadges.push(newBadge)  
   
 // Add to newly earned badges  
 newlyEarnedBadges.push(newBadge)  
   
 // Award points  
 userData.points += achievement.points  
 END IF  
 END FOREACH  
   
 // Save updated badges  
 saveToStorage("proMoodTracker:badges", userBadges)  
   
 // Save updated user data  
 saveToStorage("proMoodTracker:user", userData)  
   
 // Display notifications for new badges if any were earned  
 IF newlyEarnedBadges.length > 0 THEN  
 FOREACH badge IN newlyEarnedBadges  
 showAchievementNotification(badge)  
 END FOREACH  
   
 // Play celebration animation if configured  
 IF userPreferences.enableAchievementCelebration THEN  
 playCelebrationAnimation()  
 END IF  
 END IF  
   
 RETURN newlyEarnedBadges  
END FUNCTION

### 4.1.5 Theme Application Algorithm

Theming is an important aspect of the application’s personalization. The following pseudocode shows how themes are applied:

FUNCTION applyTheme(themeName, customTheme = null):  
 // Get predefined themes  
 predefinedThemes = {  
 "light": {  
 primary: "#6366f1",  
 secondary: "#8b5cf6",  
 background: "#ffffff",  
 paper: "#f9fafb",  
 text: "#111827",  
 moodColors: {  
 "Very Bad": "#d32f2f",  
 "Bad": "#f57c00",  
 "Okay": "#ffd600",  
 "Good": "#4caf50",  
 "Very Good": "#2196f3"  
 }  
 },  
 "dark": {  
 primary: "#818cf8",  
 secondary: "#a78bfa",  
 background: "#111827",  
 paper: "#1f2937",  
 text: "#f9fafb",  
 moodColors: {  
 "Very Bad": "#ef5350",  
 "Bad": "#ff9800",  
 "Okay": "#ffea00",  
 "Good": "#66bb6a",  
 "Very Good": "#42a5f5"  
 }  
 },  
 // Other predefined themes...  
 }  
   
 // Get custom themes  
 customThemes = getFromStorage("proMoodTracker:themes") || []  
   
 // Determine which theme to apply  
 themeToApply = null  
   
 IF customTheme IS NOT NULL THEN  
 // Apply custom theme object directly  
 themeToApply = customTheme  
 ELSE IF themeName IN predefinedThemes THEN  
 // Apply predefined theme  
 themeToApply = predefinedThemes[themeName]  
 ELSE  
 // Check if it's a custom theme by name  
 customThemeMatch = customThemes.find(theme => theme.name === themeName)  
   
 IF customThemeMatch THEN  
 themeToApply = customThemeMatch  
 ELSE  
 // Fallback to light theme if not found  
 themeToApply = predefinedThemes["light"]  
 END IF  
 END IF  
   
 // Apply theme to CSS variables  
 document.documentElement.style.setProperty('--color-primary', themeToApply.primary)  
 document.documentElement.style.setProperty('--color-secondary', themeToApply.secondary)  
 document.documentElement.style.setProperty('--color-background', themeToApply.background)  
 document.documentElement.style.setProperty('--color-paper', themeToApply.paper)  
 document.documentElement.style.setProperty('--color-text', themeToApply.text)  
   
 // Apply mood colors  
 FOR EACH mood, color IN themeToApply.moodColors  
 document.documentElement.style.setProperty('--color-mood-' + mood.toLowerCase().replace(' ', '-'), color)  
 END FOR  
   
 // Apply theme class to body  
 document.body.className = 'theme-' + (customTheme ? 'custom' : themeName)  
   
 // Check for dark mode to set meta theme-color  
 IF themeToApply.background.startsWith('#1') OR calculateLuminance(themeToApply.background) < 0.5 THEN  
 setMetaThemeColor(themeToApply.background)  
 document.body.classList.add('dark-theme')  
 ELSE  
 setMetaThemeColor(themeToApply.primary)  
 document.body.classList.remove('dark-theme')  
 END IF  
   
 // Save theme preference in settings  
 updateUserSettings({ theme: themeName })  
   
 // Return the applied theme  
 RETURN {  
 name: customTheme ? customTheme.name : themeName,  
 colors: themeToApply  
 }  
END FUNCTION

## 4.2 Use Case Diagram

Use case diagrams help visualize the main functionalities of the Pro Mood Tracker application from the user’s perspective. They illustrate the interactions between users and the system, identifying key features and their relationships.

### 4.2.1 Core System Use Case Diagram

The core system use case diagram presents an overview of the essential functions of the Pro Mood Tracker application. It demonstrates how users interact with the main features of the system.

*See Figure 4.1: Core System Use Case Diagram*

### 4.2.2 Data Management Use Case Diagram

The data management use case diagram focuses on how users interact with the data storage, export, and management features of the Pro Mood Tracker application.

*See Figure 4.2: Data Management Use Case Diagram*

### 4.2.3 Analytics Use Case Diagram

The analytics use case diagram illustrates how users interact with the data analysis and visualization features of the Pro Mood Tracker application.

*See Figure 4.3: Analytics Use Case Diagram*

## 4.3 ER Diagram

The Entity-Relationship (ER) diagram represents the logical structure of the Pro Mood Tracker application’s data model. It defines the entities, their attributes, and the relationships between them.

### 4.3.1 Conceptual Data Model

The conceptual data model provides a high-level overview of the entities and their relationships without detailing specific attributes or implementation details.

*See Figure 4.4: Conceptual Data Model ER Diagram*

### 4.3.2 Detailed Data Model

The detailed data model expands on the conceptual model, including all entity attributes, relationship cardinalities, and data types.

*See Figure 4.5: Detailed ER Diagram*

## 4.4 DFD Diagram

Data Flow Diagrams (DFD) illustrate how data moves through the Pro Mood Tracker application. They show the various processes, data stores, and external entities, along with the data flows between them.

### 4.4.1 Context Level DFD (Level 0)

The context level DFD provides a high-level view of the entire system as a single process, showing its interactions with external entities.

*See Figure 4.6: Context Level DFD*

### 4.4.2 Level 1 DFD

The Level 1 DFD expands the single process from the context diagram into the main processes of the system, showing the primary data flows between them.

*See Figure 4.7: Level 1 DFD*

### 4.4.3 Level 2 DFD for Mood Tracking

The Level 2 DFD focuses specifically on the mood tracking subsystem, breaking it down into more detailed processes.

*See Figure 4.8: Level 2 DFD for Mood Tracking*