Market Outcomes Components Guide

Overview

The GomaUI framework provides a powerful set of components for displaying betting market outcomes with a hierarchical architecture that follows MVVM principles. This guide explains how to use the three main components: `OutcomeItemView`, `MarketOutcomesLineView`, and `MarketOutcomesMultiLineView`, and how they work together.

Component Hierarchy

```
MarketOutcomesMultiLineView
├── MarketOutcomesLineView (multiple instances)
    ├── OutcomeItemView (left outcome)
    ├── OutcomeItemView (middle outcome - optional)
    └── OutcomeItemView (right outcome)
```

1. OutcomeItemView

Purpose

The `OutcomeItemView` is the foundational component that displays a single betting outcome with interactive selection, odds change animations, and accessibility support.

Key Features

- **Interactive Selection**: Tap to select/deselect with visual feedback
- **Odds Change Animations**: Regulatory-compliant animations for odds increases/decreases
- **Accessibility Support**: Full VoiceOver and accessibility trait support
- **Haptic Feedback**: Tactile feedback for user interactions
- **Customizable Styling**: Uses StyleProvider for consistent theming

Basic Usage

```swift
import GomaUI

// Create outcome data
let outcomeData = OutcomeItemData(
    id: "home",
    title: "Home",
    value: "1.85",
    oddsChangeDirection: .none,
    isSelected: false,
    isDisabled: false
)

// Create view model
let viewModel = MockOutcomeItemViewModel(outcomeData: outcomeData)

// Create and configure the view
let outcomeView = OutcomeItemView(viewModel: viewModel)

// Set up callbacks
outcomeView.onTap = { [weak self] in
    print("Outcome tapped")
    // Handle selection logic
}

outcomeView.onLongPress = { [weak self] in
    print("Outcome long pressed")
    // Handle long press actions (e.g., show details)
}

// Add to your view hierarchy
parentView.addSubview(outcomeView)
```

Customization Options

Styling

The component uses `StyleProvider` for consistent theming:

```
// Colors are automatically applied from StyleProvider
// - Primary color for selected state
// - Secondary color for disabled state
// - Text colors for labels
// - Background colors for different states
```

Odds Change Animations

```
// Update odds with automatic direction calculation
viewModel.updateValue("1.95")

// Or specify direction manually
viewModel.updateValue("1.95", changeDirection: .up)

// Clear change indicators after animation
viewModel.clearOddsChangeIndicator()
```

State Management

```
// Toggle selection
let isSelected = viewModel.toggleSelection()

// Set specific states
viewModel.setSelected(true)
viewModel.setDisabled(false)
```

Data Model

```
public struct OutcomeItemData {
    public let id: String
    public let title: String
    public let value: String
    public let oddsChangeDirection: OddsChangeDirection
    public let isSelected: Bool
    public let isDisabled: Bool
    public let previousValue: String?
    public let changeTimestamp: Date?
}
```

2. MarketOutcomesLineView

Purpose

The `MarketOutcomesLineView` displays a single line of betting outcomes (2-column or 3-column layout) and manages multiple `OutcomeItemView` instances following proper MVVM architecture.

Key Features

- **Flexible Layout**: Supports 2-column (Over/Under) and 3-column (Home/Draw/Away) layouts
- **Multiple Display Modes**: Normal outcomes, suspended markets, or "see all" states
- **Child ViewModel Management**: Parent creates and manages child view models
- **Interactive Selection**: Handles outcome selection with callbacks
- **Odds Change Forwarding**: Automatically forwards odds changes to child components

Basic Usage

```swift
import GomaUI

// Create market outcome data
let homeOutcome = MarketOutcomeData(
    id: "home",
    title: "Home",
    value: "1.85",
    isSelected: false,
    isDisabled: false
)

let drawOutcome = MarketOutcomeData(
    id: "draw",
    title: "Draw",
    value: "3.55",
    isSelected: false,
    isDisabled: false
)

let awayOutcome = MarketOutcomeData(
    id: "away",
    title: "Away",
    value: "4.20",
    isSelected: false,
    isDisabled: false
)

// Create view model
let viewModel = MockMarketOutcomesLineViewModel(
    displayMode: .triple,
    leftOutcome: homeOutcome,
    middleOutcome: drawOutcome,
    rightOutcome: awayOutcome
)

// Create and configure the view
let lineView = MarketOutcomesLineView(viewModel: viewModel)

// Set up callbacks
lineView.onOutcomeSelected = { outcomeType in
    print("Outcome selected: \(outcomeType)")
}

lineView.onOutcomeDeselected = { outcomeType in
    print("Outcome deselected: \(outcomeType)")
}

lineView.onOutcomeLongPress = { outcomeType in
    print("Outcome long pressed: \(outcomeType)")
}

lineView.onSeeAllTapped = {
    print("See all markets tapped")
}

// Add to your view hierarchy
parentView.addSubview(lineView)
```

Display Modes

Normal Markets

```
// Two-way market (Over/Under)
let twoWayViewModel = MockMarketOutcomesLineViewModel(
    displayMode: .double,
    leftOutcome: overOutcome,
    rightOutcome: underOutcome
)

// Three-way market (Home/Draw/Away)
let threeWayViewModel = MockMarketOutcomesLineViewModel(
    displayMode: .triple,
    leftOutcome: homeOutcome,
    middleOutcome: drawOutcome,
    rightOutcome: awayOutcome
)
```

Special States

```
// Suspended market
let suspendedViewModel = MockMarketOutcomesLineViewModel(
    displayMode: .suspended(text: "Market Suspended")
)

// See all markets
let seeAllViewModel = MockMarketOutcomesLineViewModel(
    displayMode: .seeAll(text: "See All Markets")
)
```

MVVM Architecture

The `MarketOutcomesLineView` follows proper MVVM principles where the parent view model creates child view models:

```
// The view model protocol includes child creation
public protocol MarketOutcomesLineViewModelProtocol {
    // ... other methods ...

    // Parent creates child view models
    func createOutcomeViewModel(for outcomeType: OutcomeType) -> OutcomeItemViewModelPr
}
```

This ensures:

- **Proper ownership**: Parent view model owns child view models
- **Data flow**: Parent manages data and forwards to children
- **Event handling**: Parent coordinates events between children
- **Testability**: Easy to inject different implementations

Customization Options

Updating Odds

```swift
// Update odds with automatic change detection
viewModel.updateOddsValue(type: .left, newValue: "1.95")

// Manual direction specification
viewModel.updateOddsValue(type: .left, value: "1.95", changeDirection: .up)

// Clear change indicators
viewModel.clearOddsChangeIndicator(type: .left)
```

Selection Management

```swift
// Toggle outcome selection
let isSelected = viewModel.toggleOutcome(type: .left)
```

Display Mode Changes

```swift
// Change display mode dynamically
viewModel.setDisplayMode(.suspended(text: "Temporarily Unavailable"))
```

Data Models

```swift
public struct MarketOutcomeData {
    public let id: String
    public let title: String
    public let value: String
    public let oddsChangeDirection: OddsChangeDirection
    public let isSelected: Bool
    public let isDisabled: Bool
    public let previousValue: String?
    public let changeTimestamp: Date?
}

public enum MarketDisplayMode {
    case triple  // Three-way market
    case double  // Two-way market
    case suspended(text: String)
    case seeAll(text: String)
}

public enum OutcomeType {
    case left
    case middle
    case right
}
```

3. MarketOutcomesMultiLineView

Purpose

The `MarketOutcomesMultiLineView` displays multiple market outcome lines in a vertical layout, perfect for market groups where you have multiple related markets (e.g., Over/Under 1.5, 2.5, 3.5).

Key Features

- **Multiple Market Lines**: Display multiple outcome lines in a vertical stack
- **Mixed Layout Support**: Each line can independently be 2-column or 3-column
- **Independent Line States**: Each line can be suspended, disabled, or active independently
- **Group Management**: Optional group title and coordinated state management
- **Scalable Architecture**: Uses `MarketOutcomesLineView` instances internally

Basic Usage

```
import GomaUI

// Create market group data
let marketGroup = MarketGroupData(
    id: "over_under_goals",
    title: "Total Goals",
    marketLines: [
        // Over/Under 1.5
        MarketLineData(
            id: "goals_1_5",
            leftOutcome: MarketOutcomeData(id: "over_1_5", title: "Over 1.5", value: "1
            rightOutcome: MarketOutcomeData(id: "under_1_5", title: "Under 1.5", value:
            displayMode: .double,
            lineType: .overUnder,
            isLineDisabled: false
        ),
        // Over/Under 2.5
        MarketLineData(
            id: "goals_2_5",
            leftOutcome: MarketOutcomeData(id: "over_2_5", title: "Over 2.5", value: "1
            rightOutcome: MarketOutcomeData(id: "under_2_5", title: "Under 2.5", value:
            displayMode: .double,
            lineType: .overUnder,
            isLineDisabled: false
        ),
        // Over/Under 3.5
        MarketLineData(
            id: "goals_3_5",
            leftOutcome: MarketOutcomeData(id: "over_3_5", title: "Over 3.5", value: "3
            rightOutcome: MarketOutcomeData(id: "under_3_5", title: "Under 3.5", value:
            displayMode: .double,
            lineType: .overUnder,
            isLineDisabled: false
        )
    ]
)

// Create view model
let viewModel = MockMarketOutcomesMultiLineViewModel.overUnderMarketGroup

// Create and configure the view
```

```swift
let multiLineView = MarketOutcomesMultiLineView(viewModel: viewModel)

// Set up callbacks
multiLineView.onOutcomeSelected = { lineId, outcomeType in
    print("Outcome selected in line \(lineId): \(outcomeType)")
}

multiLineView.onOutcomeDeselected = { lineId, outcomeType in
    print("Outcome deselected in line \(lineId): \(outcomeType)")
}

multiLineView.onLineLongPress = { lineId in
    print("Line long pressed: \(lineId)")
}

// Add to your view hierarchy
parentView.addSubview(multiLineView)
```

Mixed Layout Support

```swift
// Create a market group with mixed layouts
let mixedMarketGroup = MarketGroupData(
    id: "mixed_markets",
    title: "Popular Markets",
    marketLines: [
        // 3-column layout (Home/Draw/Away)
        MarketLineData(
            id: "match_result",
            leftOutcome: MarketOutcomeData(id: "home", title: "Home", value: "1.85"),
            middleOutcome: MarketOutcomeData(id: "draw", title: "Draw", value: "3.55"),
            rightOutcome: MarketOutcomeData(id: "away", title: "Away", value: "4.20"),
            displayMode: .triple,
            lineType: .matchResult,
            isLineDisabled: false
        ),
        // 2-column layout (Over/Under)
        MarketLineData(
            id: "total_goals",
            leftOutcome: MarketOutcomeData(id: "over", title: "Over 2.5", value: "1.95"
            rightOutcome: MarketOutcomeData(id: "under", title: "Under 2.5", value: "1.
            displayMode: .double,
            lineType: .overUnder,
            isLineDisabled: false
        )
    ]
)
```

Independent Line Management

```swift
// Suspend a specific line
viewModel.suspendLine(lineId: "goals_2_5", reason: "Temporarily unavailable")

// Resume a suspended line
viewModel.resumeLine(lineId: "goals_2_5")

// Disable/enable a line
viewModel.setLineDisabled(lineId: "goals_3_5", disabled: true)

// Update odds for a specific outcome in a specific line
viewModel.updateOddsValue(lineId: "goals_1_5", outcomeType: .left, newValue: "1.30")
```

Customization Options

Group Title

```swift
// Show/hide group title
viewModel.setGroupTitleVisible(true)

// Update group title
viewModel.updateGroupTitle("Updated Market Group")
```

Loading and Error States

```swift
// Show loading state
viewModel.setLoadingState(true)

// Show error state
viewModel.setErrorState("Failed to load markets")

// Clear error state
viewModel.clearErrorState()
```

Line Ordering

```
// Reorder lines
viewModel.reorderLines(newOrder: ["goals_3_5", "goals_2_5", "goals_1_5"])

// Add new line
viewModel.addLine(newLineData)

// Remove line
viewModel.removeLine(lineId: "goals_1_5")
```

Data Models

```
public struct MarketGroupData {
    public let id: String
    public let title: String?
    public let marketLines: [MarketLineData]
}

public struct MarketLineData {
    public let id: String
    public let leftOutcome: MarketOutcomeData?
    public let middleOutcome: MarketOutcomeData?
    public let rightOutcome: MarketOutcomeData?
    public let displayMode: MarketDisplayMode
    public let lineType: MarketLineType
    public let isLineDisabled: Bool
}

public enum MarketLineType {
    case matchResult      // 1X2
    case overUnder       // Over/Under
    case doubleChance    // 1X, 12, X2
    case handicap        // Asian Handicap
    case custom(String)  // Custom market type
}
```

Component Relationships

Data Flow

1. **Top-Down**: `MarketOutcomesMultiLineView` → `MarketOutcomesLineView` → `OutcomeItemView`
2. **Event Bubbling**: User interactions bubble up through callbacks
3. **State Synchronization**: Parent components manage child state through view models

MVVM Architecture

```
MarketOutcomesMultiLineViewModel
├── Creates multiple MarketOutcomesLineViewModel instances
    ├── Each creates OutcomeItemViewModel instances
    └── Manages data flow and event coordination
```

Communication Patterns

Parent to Child (Data Flow)

```swift
// Parent view model creates child view models
func createLineViewModel(for lineData: MarketLineData) -> MarketOutcomesLineViewModelPr

// Parent forwards data updates to children
func updateLineData(lineId: String, newData: MarketLineData)
```

Child to Parent (Event Bubbling)

```swift
// Child events bubble up through callbacks
multiLineView.onOutcomeSelected = { lineId, outcomeType in
    // Handle at parent level
}

lineView.onOutcomeSelected = { outcomeType in
    // Forward to multi-line parent
}
```

Best Practices

1. View Model Lifecycle

```swift
// Always clean up when reusing views
lineView.cleanupForReuse()

// Proper view model creation
let viewModel = parentViewModel.createChildViewModel(for: data)
```

2. State Management

```swift
// Use reactive patterns for state updates
viewModel.marketStatePublisher
    .receive(on: DispatchQueue.main)
    .sink { state in
        // Update UI
    }
    .store(in: &cancellables)
```

3. Error Handling

```swift
// Handle loading and error states
multiLineView.onLoadingStateChanged = { isLoading in
    // Show/hide loading indicator
}

multiLineView.onErrorOccurred = { error in
    // Display error message
}
```

4. Accessibility

```swift
// Components automatically provide accessibility support
// Customize if needed:
outcomeView.accessibilityLabel = "Custom label"
outcomeView.accessibilityHint = "Custom hint"
```

5. Performance

```swift
// Use appropriate cleanup for table/collection view cells
override func prepareForReuse() {
    super.prepareForReuse()
    marketLineView.cleanupForReuse()
}
```

Testing

Unit Testing View Models

```swift
func testOutcomeSelection() {
    let viewModel = MockOutcomeItemViewModel.homeOutcome

    let isSelected = viewModel.toggleSelection()

    XCTAssertTrue(isSelected)
    // Verify state changes
}
```

Integration Testing

```swift
func testMarketLineInteraction() {
    let lineViewModel = MockMarketOutcomesLineViewModel.threeWayMarket
    let lineView = MarketOutcomesLineView(viewModel: lineViewModel)

    var selectedOutcome: OutcomeType?
    lineView.onOutcomeSelected = { outcome in
        selectedOutcome = outcome
    }

    // Simulate user interaction
    lineViewModel.toggleOutcome(type: .left)

    XCTAssertEqual(selectedOutcome, .left)
}
```

Conclusion

The Market Outcomes components provide a flexible, scalable architecture for displaying betting markets with proper MVVM separation, reactive state management, and comprehensive customization options. The hierarchical design allows for complex market displays while maintaining clean code organization and testability.

For more specific implementation details, refer to the individual component documentation and example projects in the GomaUI framework.