Daniel Jochum

CPSC 411

Prof. James Shen

14 December 2025

Pick For Me

CPSC 411 Final Project Report

Pick For Me is a project created as a final project for CPSC 411: Mobile App Development. It is designed to demonstrate SwiftUI proficiency and include many things learned throughout the semester.

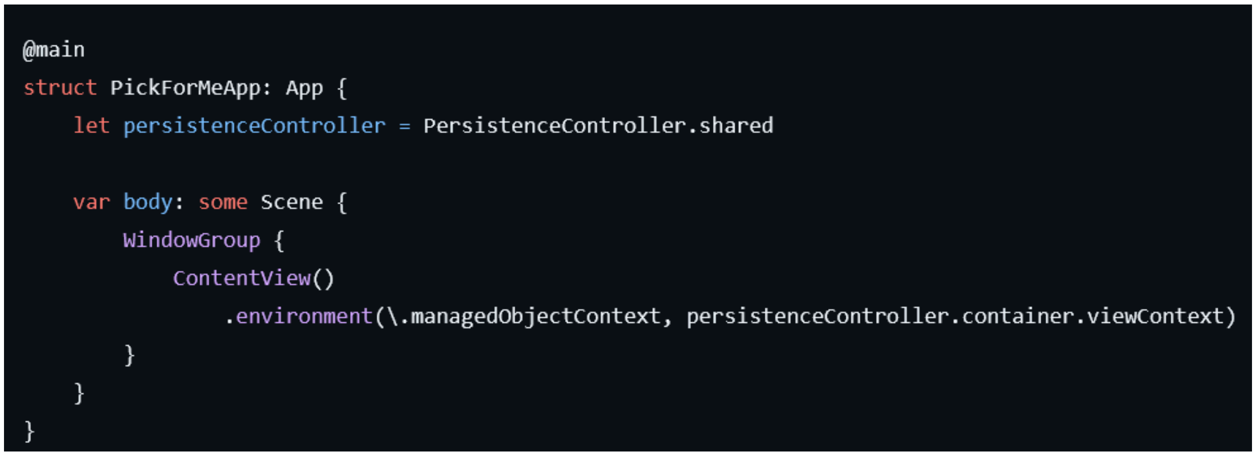
Pick For Me has four tabs. The first is the Items Tab. In this tab one can add a number of items to a list. Users can also manage this list by removing single entries, or deleting all of them at once. The next two tabs utilize this list of items in their functions.

Tab number 2, the Wheel Tab, places these items on a wheel. The user can spin this wheel to randomly select one of the options inputted.

The third tab is the Bracket Tab. This tab places items in a tournament-style bracket, asking the user to repeatedly choose between two of them until only the champion remains.

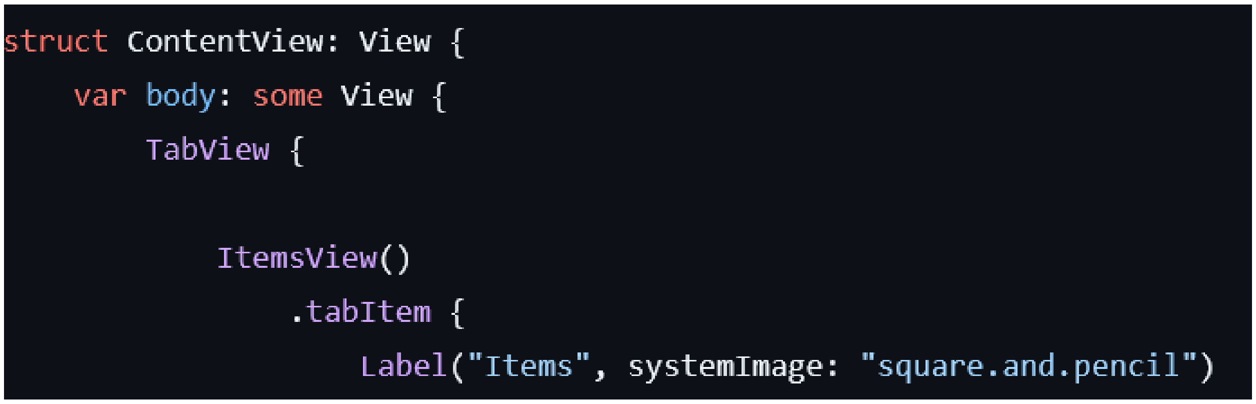
The fourth and final tab contains a handful of small random-related functions. It holds a d6, d20, d100, a coin-flip, and a random number generator.

Main lies in PickForMeApp.swift, and calls ContentView().



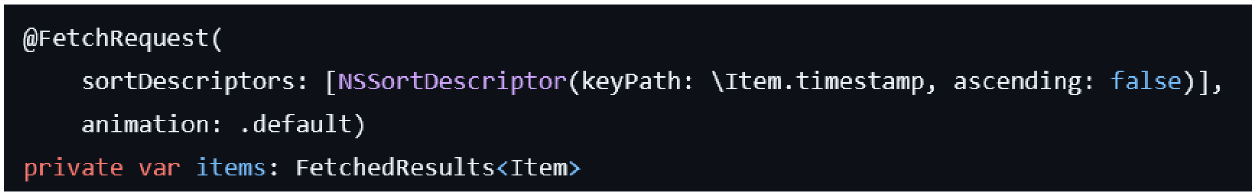
```swift
@main
struct PickForMeApp: App {
    let persistenceController = PersistenceController.shared

    var body: some Scene {
        WindowGroup {
            ContentView()
                .environment(\.managedObjectContext, persistenceController.container.viewContext)
        }
    }
}
```

ContentView.swift contains a TabView with links to our four tabs.



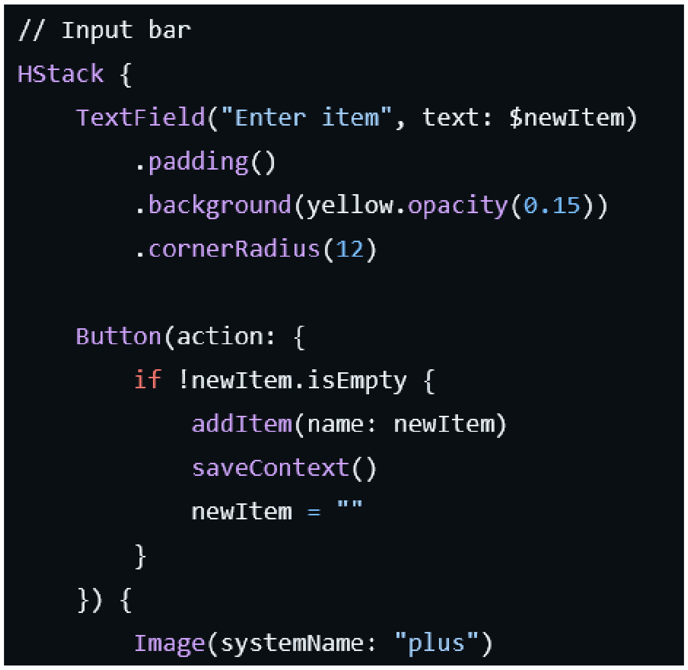
```swift
struct ContentView: View {
    var body: some View {
        TabView {

            ItemsView()
                .tabItem {
                    Label("Items", systemImage: "square.and.pencil")
```

ItemsView.swift houses our first tab. It uses CoreData to store all of the inputted items for future use and use by other tabs.



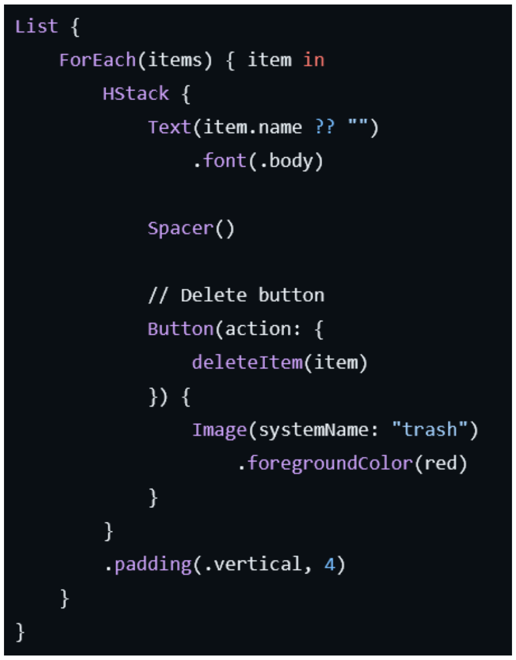
```swift
@FetchRequest(
    sortDescriptors: [NSSortDescriptor(keyPath: \Item.timestamp, ascending: false)],
    animation: .default)
private var items: FetchedResults<Item>
```

The top of the Items Tab is an entry field and a button that adds the entry to a list.



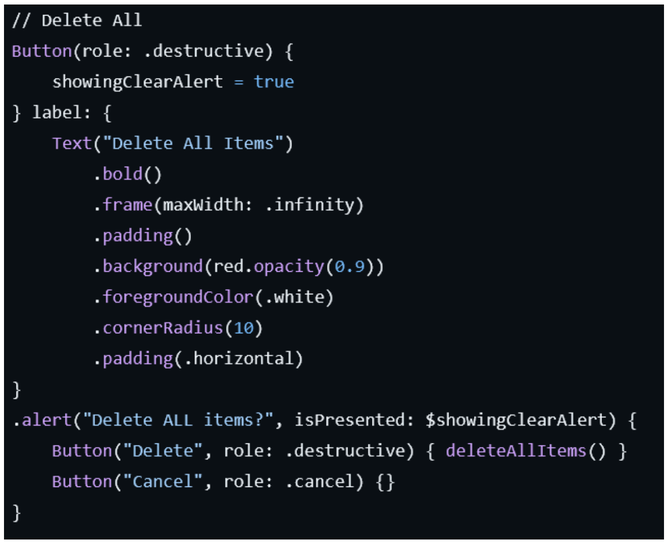
```
// Input bar
HStack {
    TextField("Enter item", text: $newItem)
        .padding()
        .background(yellow.opacity(0.15))
        .cornerRadius(12)

    Button(action: {
        if !newItem.isEmpty {
            addItem(name: newItem)
            saveContext()
            newItem = ""
        }
    }) {
        Image(systemName: "plus")
```

Then is the list of items. Each list entry has the name of the item and a delete button.



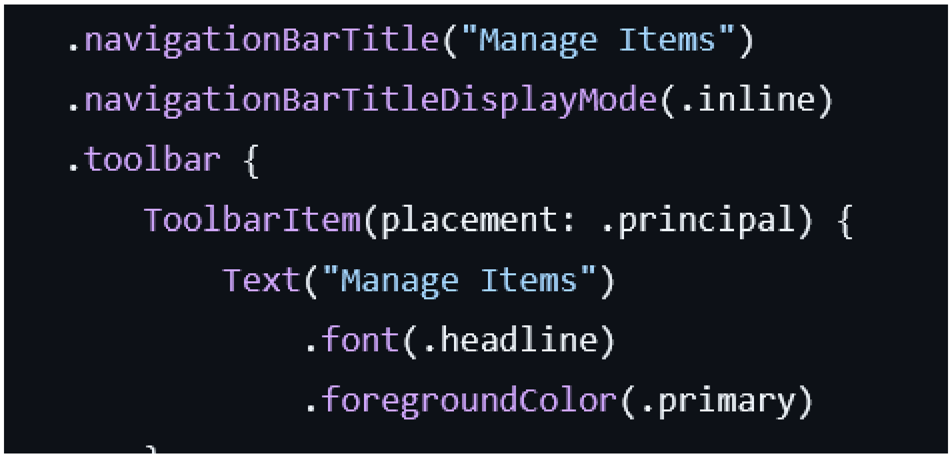
```
List {
    ForEach(items) { item in
        HStack {
            Text(item.name ?? "")
                .font(.body)

            Spacer()

            // Delete button
            Button(action: {
                deleteItem(item)
            }) {
                Image(systemName: "trash")
                    .foregroundColor(red)
            }
        }
        .padding(.vertical, 4)
    }
}
```

At the bottom of the screen is a Delete All button, which removes all list entries.



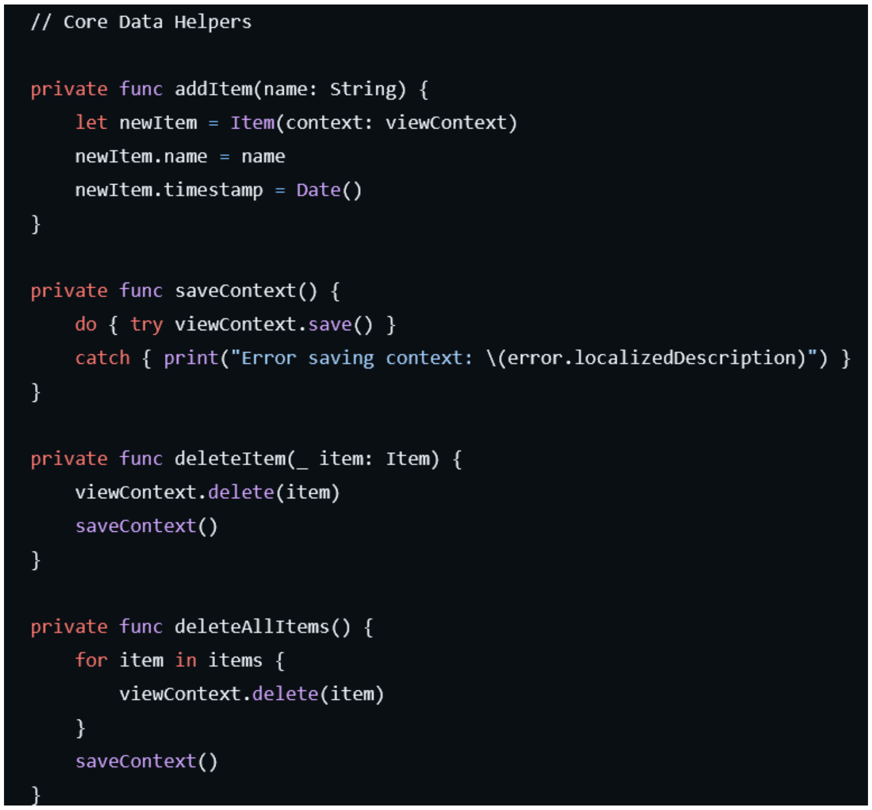
```swift
// Delete All
Button(role: .destructive) {
    showingClearAlert = true
} label: {
    Text("Delete All Items")
        .bold()
        .frame(maxWidth: .infinity)
        .padding()
        .background(red.opacity(0.9))
        .foregroundColor(.white)
        .cornerRadius(10)
        .padding(.horizontal)
}
.alert("Delete ALL items?", isPresented: $showingClearAlert) {
    Button("Delete", role: .destructive) { deleteAllItems() }
    Button("Cancel", role: .cancel) {}
}
```

Our tab has a title at the top. All other tabs also use this same structure.



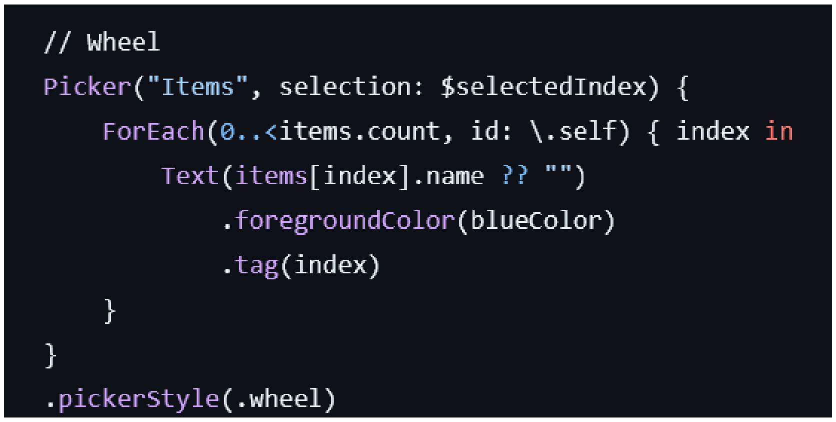
```swift
.navigationBarTitle("Manage Items")
.navigationBarTitleDisplayMode(.inline)
.toolbar {
    ToolbarItem(placement: .principal) {
        Text("Manage Items")
            .font(.headline)
            .foregroundColor(.primary)
```

The last functions of note in this tab are its Core Data Helpers. These functions are used by our above buttons to save data to CoreData.



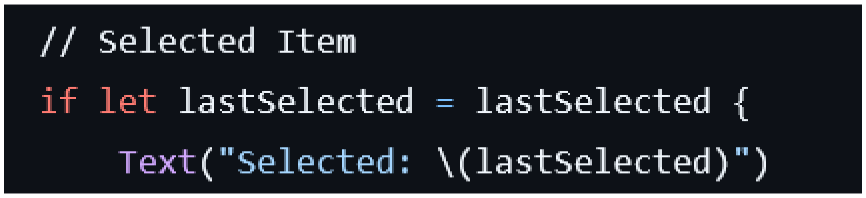
```
// Core Data Helpers

private func addItem(name: String) {
    let newItem = Item(context: viewContext)
    newItem.name = name
    newItem.timestamp = Date()
}

private func saveContext() {
    do { try viewContext.save() }
    catch { print("Error saving context: \(error.localizedDescription)") }
}

private func deleteItem(_ item: Item) {
    viewContext.delete(item)
    saveContext()
}

private func deleteAllItems() {
    for item in items {
        viewContext.delete(item)
    }
    saveContext()
}
```

Moving on to the next tab, WheelView.swift, it also imports items from CoreData reusing the same code (not reposted here). The top of the Wheel Tab has a picker wheel.



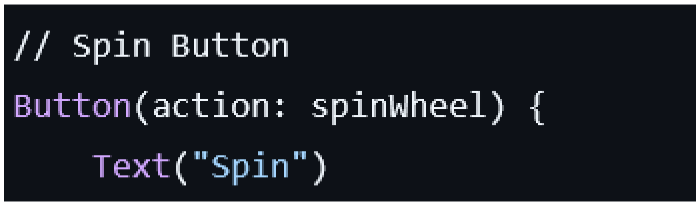
```
// Wheel
Picker("Items", selection: $selectedIndex) {
    ForEach(0..<items.count, id: \.self) { index in
        Text(items[index].name ?? "")
            .foregroundColor(blueColor)
            .tag(index)
    }
}
.pickerStyle(.wheel)
```

Below is text showcasing the selected item, if one has been selected.



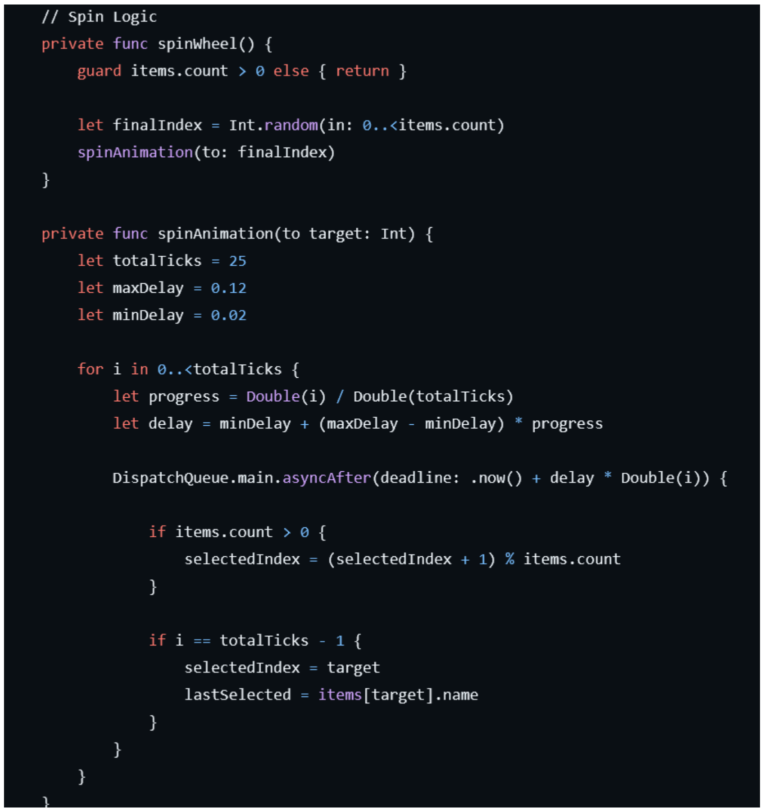
```
// Selected Item
if let lastSelected = lastSelected {
    Text("Selected: \(lastSelected)")
```

Last on this tab is the spin button.



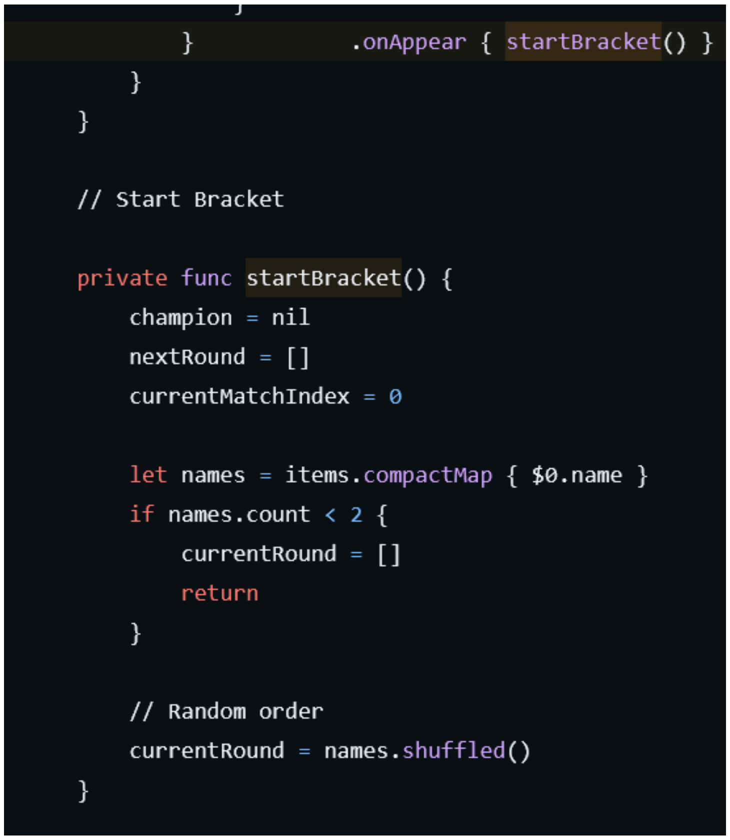
```
// Spin Button
Button(action: spinWheel) {
    Text("Spin")
```

The logic for the spin button starts by choosing a random item, then calling the "animation". The animation simply works by selecting successive items in the wheel at an increasing delay using a DispatchQueue.



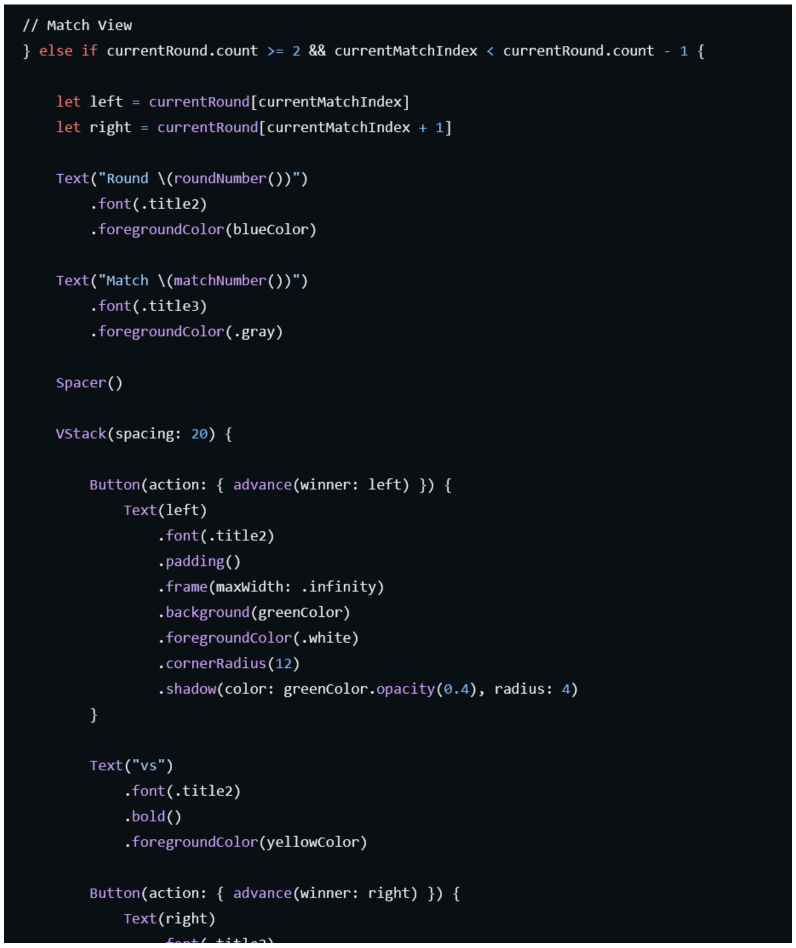
```
// Spin Logic
private func spinWheel() {
    guard items.count > 0 else { return }

    let finalIndex = Int.random(in: 0..<items.count)
    spinAnimation(to: finalIndex)
}

private func spinAnimation(to target: Int) {
    let totalTicks = 25
    let maxDelay = 0.12
    let minDelay = 0.02

    for i in 0..<totalTicks {
        let progress = Double(i) / Double(totalTicks)
        let delay = minDelay + (maxDelay - minDelay) * progress

        DispatchQueue.main.asyncAfter(deadline: .now() + delay * Double(i)) {

            if items.count > 0 {
                selectedIndex = (selectedIndex + 1) % items.count
            }

            if i == totalTicks - 1 {
                selectedIndex = target
                lastSelected = items[target].name
            }
        }
    }
}
```

The third tab, BracketView.swift has an onAppear function called startBracket(). This sets up the tournament bracket by loading the items into a shuffled array.



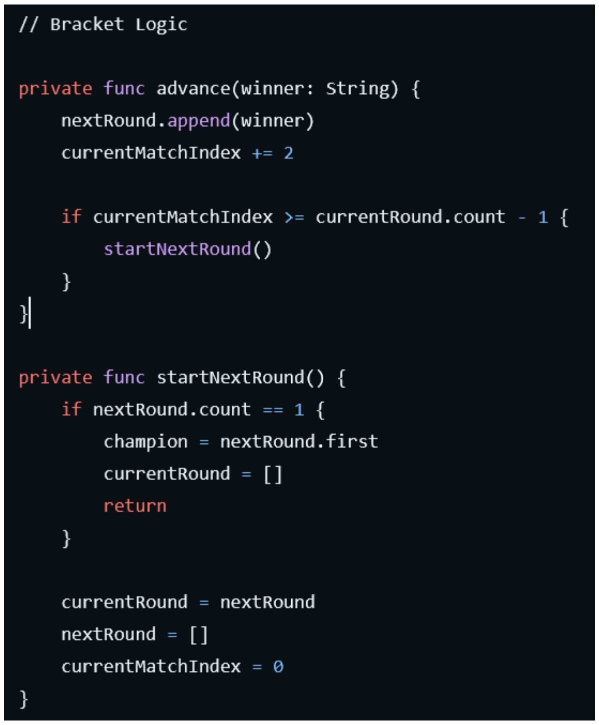
```
            }           .onAppear { startBracket() }
        }
    }

    // Start Bracket

    private func startBracket() {
        champion = nil
        nextRound = []
        currentMatchIndex = 0

        let names = items.compactMap { $0.name }
        if names.count < 2 {
            currentRound = []
            return
        }

        // Random order
        currentRound = names.shuffled()
    }
```

The Bracket Tab displays a current round and match number, and two buttons with different items for the user to choose from.



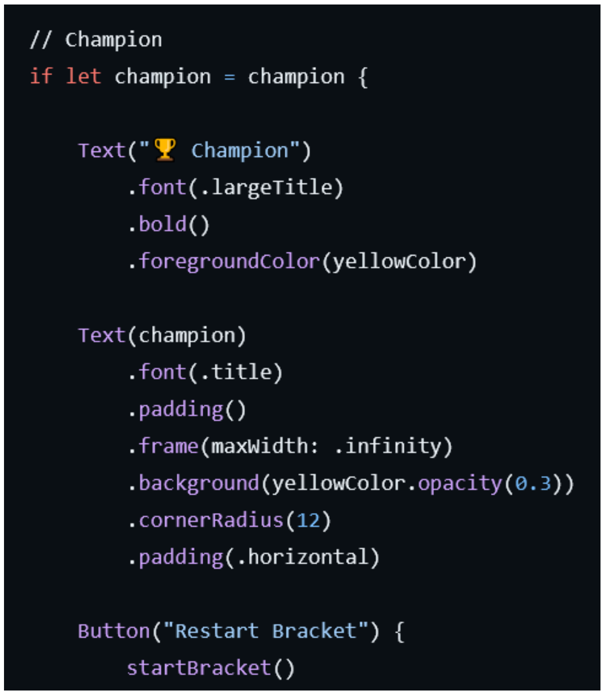
```
// Match View
} else if currentRound.count >= 2 && currentMatchIndex < currentRound.count - 1 {

    let left = currentRound[currentMatchIndex]
    let right = currentRound[currentMatchIndex + 1]

    Text("Round \(roundNumber())")
        .font(.title2)
        .foregroundColor(blueColor)

    Text("Match \(matchNumber())")
        .font(.title3)
        .foregroundColor(.gray)

    Spacer()

    VStack(spacing: 20) {

        Button(action: { advance(winner: left) }) {
            Text(left)
                .font(.title2)
                .padding()
                .frame(maxWidth: .infinity)
                .background(greenColor)
                .foregroundColor(.white)
                .cornerRadius(12)
                .shadow(color: greenColor.opacity(0.4), radius: 4)
        }

        Text("vs")
            .font(.title2)
            .bold()
            .foregroundColor(yellowColor)

        Button(action: { advance(winner: right) }) {
            Text(right)
```

Each button calls an advance() function for the selected item, which adds to an array for the next round. Once each item has been picked, the function starts a new round by recursively calling a startNewRound() function, using the nextRound array as the new currentRound array.



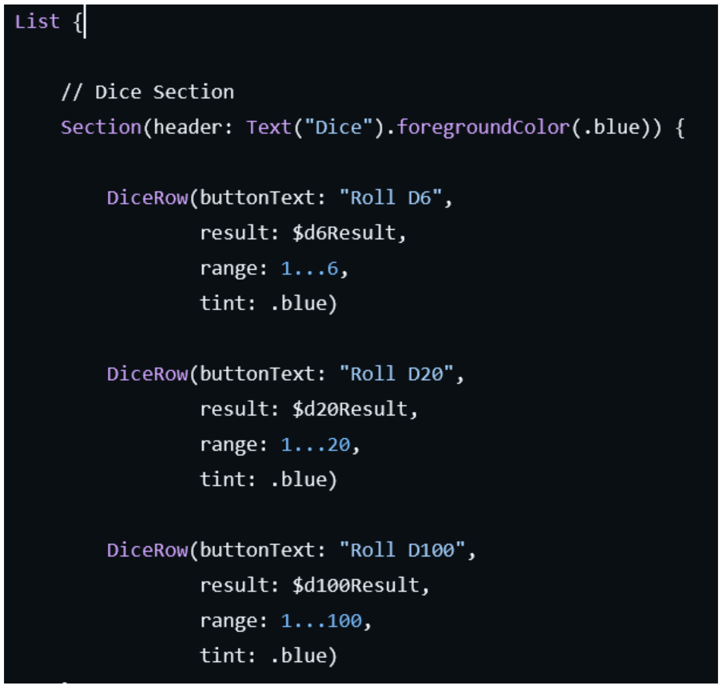
```
// Bracket Logic

private func advance(winner: String) {
    nextRound.append(winner)
    currentMatchIndex += 2

    if currentMatchIndex >= currentRound.count - 1 {
        startNextRound()
    }
}

private func startNextRound() {
    if nextRound.count == 1 {
        champion = nextRound.first
        currentRound = []
        return
    }

    currentRound = nextRound
    nextRound = []
    currentMatchIndex = 0
}
```

If a new round is called, and there is only one item remaining, the item is declared the champion. The match and round text as well as the item buttons are replaced with championship text and a new button that gives the user the option to restart the bracket.



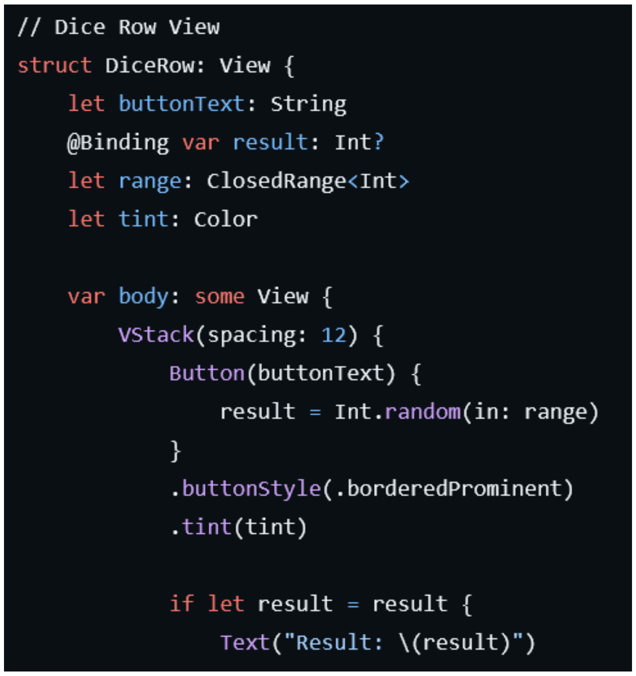
```
// Champion
if let champion = champion {

    Text("🏆 Champion")
        .font(.largeTitle)
        .bold()
        .foregroundColor(yellowColor)

    Text(champion)
        .font(.title)
        .padding()
        .frame(maxWidth: .infinity)
        .background(yellowColor.opacity(0.3))
        .cornerRadius(12)
        .padding(.horizontal)

    Button("Restart Bracket") {
        startBracket()
```

The last tab, ExtrasView.swift, has a list of several smaller functions, starting with a few dice rolling functions.



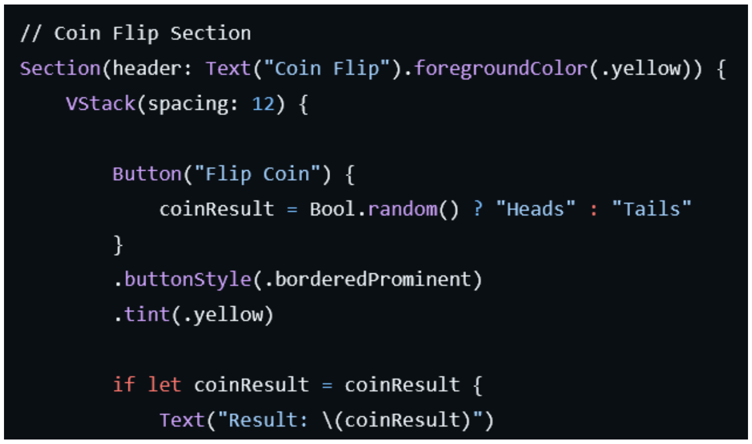
```
List {

    // Dice Section
    Section(header: Text("Dice").foregroundColor(.blue)) {

        DiceRow(buttonText: "Roll D6",
                result: $d6Result,
                range: 1...6,
                tint: .blue)

        DiceRow(buttonText: "Roll D20",
                result: $d20Result,
                range: 1...20,
                tint: .blue)

        DiceRow(buttonText: "Roll D100",
                result: $d100Result,
                range: 1...100,
                tint: .blue)
```

The dice functions use DiceRow, a custom struct that has a button and automatically updates text with a new result when clicked



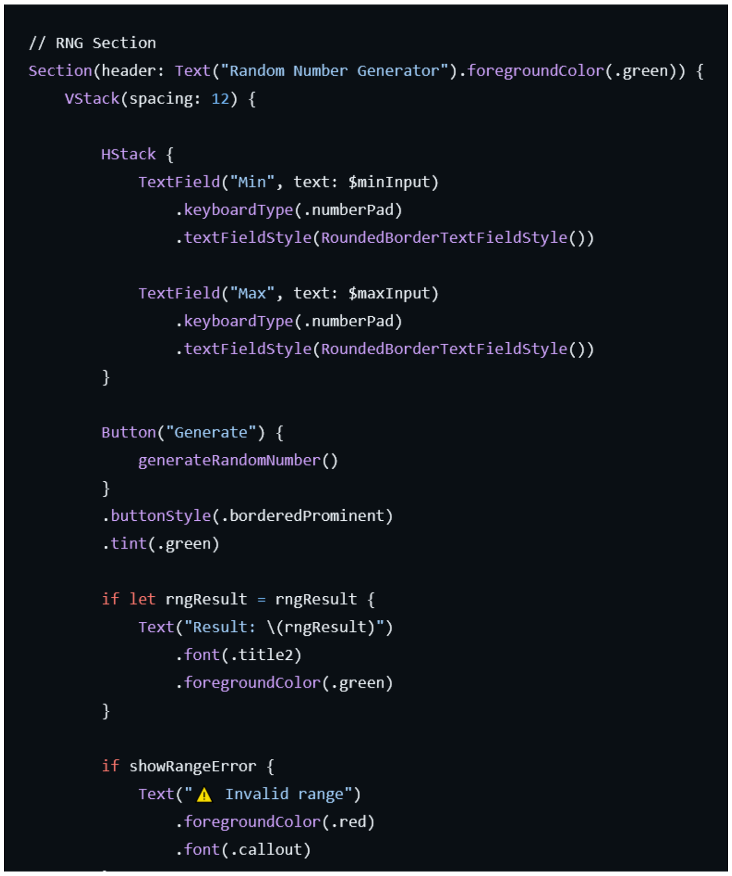
```
// Dice Row View
struct DiceRow: View {
    let buttonText: String
    @Binding var result: Int?
    let range: ClosedRange<Int>
    let tint: Color

    var body: some View {
        VStack(spacing: 12) {
            Button(buttonText) {
                result = Int.random(in: range)
            }
            .buttonStyle(.borderedProminent)
            .tint(tint)

            if let result = result {
                Text("Result: \(result)")
```

The next function is a coin-flip button that simply chooses heads or tails and pushes it to text.



```
// Coin Flip Section
Section(header: Text("Coin Flip").foregroundColor(.yellow)) {
    VStack(spacing: 12) {

        Button("Flip Coin") {
            coinResult = Bool.random() ? "Heads" : "Tails"
        }
        .buttonStyle(.borderedProminent)
        .tint(.yellow)

        if let coinResult = coinResult {
            Text("Result: \(coinResult)")
```

The final function is a random number generator. It has two text fields, for a minimum and maximum user-set range, and also a button to generate the number. There is basic error handling to make sure that the min range is lower than the max range.



```
// RNG Section
Section(header: Text("Random Number Generator").foregroundColor(.green)) {
    VStack(spacing: 12) {

        HStack {
            TextField("Min", text: $minInput)
                .keyboardType(.numberPad)
                .textFieldStyle(RoundedBorderTextFieldStyle())

            TextField("Max", text: $maxInput)
                .keyboardType(.numberPad)
                .textFieldStyle(RoundedBorderTextFieldStyle())
        }

        Button("Generate") {
            generateRandomNumber()
        }
        .buttonStyle(.borderedProminent)
        .tint(.green)

        if let rngResult = rngResult {
            Text("Result: \(rngResult)")
                .font(.title2)
                .foregroundColor(.green)
        }

        if showRangeError {
            Text("⚠️ Invalid range")
                .foregroundColor(.red)
                .font(.callout)
```

All that remains is code automatically generated by XCode. The full project code is available at my github:

https://github.com/drjochum02/CPSC-411/tree/main/PickForMe.xcodeproj

To run it, one can simply download the project files, then open and run it in XCode. The app was tested in iOS 26.1 on an iPhone 17 emulator.

Thank you for reviewing my project.