Reflection Report: Chess Armies Assignment

Ebube Esor 1253613

In this assignment, I implemented a "Peaceful Armies of Queens" program using the (file: `queenarmies.adb`), that places `m` black and `m` white queens on an `n x n` chessboard in an arrangment where no queen attacks another of a different color. Using the `ShowBoard` package (files: `showboard.ads` and `showboard.adb`) to vizualize the chessboard and it's pieces with input validation to ensure proper ranges. The program shows up to two distinct solutions for parameters parsed.

I used a **backtracking algorithm**, for the queen placement for this solution:

- **Data Structures**: I defined arrays (`Row_Array` and `Diag_Array`) to track the number of black and white queens in each row, column, and diagonal (both forward and backward).
- **Recursive Solving**: The `Solve` procedure recursively places queens, alternating between black (`1`) and white (`2`) based on the count `K`. It checks if a position is safe and no queen of the opposite color threatens it on rows, columns, or diagonals.
- **Visualization**: The `ShowBoard` package displays the board using Unicode characters (`♛` for black queens, `♕` for white queens) and a grid layout with borders for aesthetic appeal.

The program starts by prompting the user for `n` (board size, 1-10) and `m` (number of queens per color, 1-4), validates these inputs, and then initiates the solving process.

Challenges

The hardest parts for implementation were:

1. **Queen Placement Constraints**: Making sure queens of different colors do not attack each other, while allowing same-color queens to required tricky constraint management.
2. **Input Validation**: Handling invalid user inputs while avoiding runtime errors.
3. **Board Visualization**: Making an aligned chessboard display using Unicode characters was a challenge, especially with varying board sizes.
4. **Solution Distinctness**: Identifying and storing only distinct solutions.

To tackle this:

- **Constraint Management**: I used separate arrays for black and white queens (`Num_Black_Row`, `Num_White_Row`, etc.) to track their positions. Before placing a queen, I checked that no queen of the opposite color occupied the same row, column, or diagonal, using the arrays as lookups.
- **Input Validation**: I implemented checks after each input (`Get(N)` and `Get(M)`), displaying error messages (e.g., "Invalid n. Must be between 1 and 10.") and terminating the program if the inputs were out of range.
- **Visualization**: In `ShowBoard`, I used fixed `Cell_Width` of 3 characters and a helper procedure `Put_Horiz_Line` to print consistent borders. Queens were padded with spaces (e.g., " ♛ ") to align with empty cells (" "), and Unicode border characters (e.g., `┌`, `┴`) created a neat grid.
- **Distinct Solutions**: My `Boards_Equal` function compares two boards up to size `n`, and I only stored a second solution if it differed from the first, capping the output at two distinct solutions.

After completing Peaceful Chess Armies I got better at programming in ada as well as problem solving. The process of designing algorithms visualising my results excercised my planning, testing, and iteration in software development. I now feel more confident tackling similar combinatorial problems using Ada.