Objective 1: Migrate Game Logic to Blazor
Key Result 1.1: Migrate existing game logic classes to the Blazor project.
Keep the Card, Deck, Player, and AIPlayer classes in the "Models" folder without changes.
Rename the GameManager class to GameService and move it to the "Services" folder.
Sample:

```
Services
└──GameService.cs
Pages
|  Index.razor
|  Game.razor
|  Rules.razor
|
└──Components
   |  MainLayout.razor
   |  GameBoard.razor
   |  PlayerHand.razor
   |  DiscardPile.razor
   |  GameControls.razor
...
```

In Game.razor, inject GameService like this:
@inject GameService GameService
This organizational method ensures that not only the code but also CSS can be reused efficiently.
Key Result 1.2: Update the GameService class to fit Blazor's asynchronous programming model.
Convert the PlayGame method into an asynchronous method, using Task and await keywords.
Change the GetPlayerMove method to an asynchronous method, returning Task<PlayerAction>.
Update other related methods to support asynchronous operations, enabling simultaneous operations of Blazor pages and game logic.

Objective 2: Implement Blazor Game User Interface
Key Result 2.1: Create main game components.
Develop MainLayout.razor to define the overall layout of the game.
Build GameBoard.razor to display the main game interface, player hands, and discard pile.
Create PlayerHand.razor to show the player's hand.
Design DiscardPile.razor to show the discard pile.
Develop GameControls.razor to display game control buttons such as play, pass, discard, restart, etc.
Key Result 2.2: Implement game controls and player interaction.
Add buttons like "Play", "Pass", "Discard", "Restart", and "View Rules" in GameControls.razor.
In PlayerHand.razor, add "Play" and "Discard" buttons for each card.
Use Blazor's event binding and callback mechanisms to handle player actions.
Key Result 2.3: Create a game rules component.
Develop GameRules.razor to display the game rules.
Add rule content in GameRules.razor and style it appropriately.

Objective 3: Implement Game State Management and Updates
Key Result 3.1: Use Blazor's state management mechanisms to store and update game state.
Add properties in the GameService class to represent game states like the current player, player hands, discard pile, etc.
Use Blazor's EventCallback and StateHasChanged methods to notify components of changes in the game state.
Key Result 3.2: Implement real-time updates of game states.
Immediately update relevant components to reflect changes in game state after player actions such as playing, passing, or discarding.
Use Blazor's data-binding mechanisms to automatically update component displays.
Add suitable animations and transitions for player actions like playing, passing, or discarding.
Enhance the visual appeal of game components using CSS and images.

Game Components Overview:

GameService.cs
Update methods like PlayGame and GetPlayerMove to support asynchronous operations.
Add properties to represent game states, such as the current player, player hands, and the discard pile.
MainLayout.razor
Define the overall layout of the game, including the game board, player hands, discard pile, and game control buttons.
GameBoard.razor
Display the game board, player hands, and discard pile.
Use data binding to automatically update the display content in components.
PlayerHand.razor
Show the player's hand and add "Play" and "Discard" buttons for each card.
Handle player actions using event bindings and callbacks.
DiscardPile.razor
Display the cards in the discard pile.
GameControls.razor
Display game control buttons, such as "Play", "Pass", "Discard", "Restart", and "View Rules".
Manage player actions using event bindings and callbacks.
GameRules.razor
Display the game rules and set appropriate styles.

Potential Functions and Features:
Razor Components:
Define UI components like GameBoard, PlayerHand, and DiscardPile using .razor files.
Write component-specific C# code in the @code block to handle user interactions and game logic.
Implement two-way data binding with the @bind directive to synchronize UI and game state.
Event Handling:
Use event directives like @onclick and @onchange to handle user clicks, selections, and other actions.
Update game state in event handling methods and trigger UI re-rendering.
Dependency Injection:
Inject services such as GameService with the @inject directive to access game state and logic.
Use injected services in component constructors or the @code block to perform game operations.
Routing:
Define component routing with the @page directive for URLs like "/game" and "/rules".
Create navigation links with <NavLink> components to switch between different game pages.
State Management:
Inject state management services like GameState with the @inject directive.
Store and update game state using state management services, sharing state across components.
Lifecycle Methods:
Use lifecycle methods like OnInitialized() and OnParametersSet() for component initialization and data loading.
Subscribe to events and initialize game state in these methods.
Conditional Rendering:
Render different UI elements conditionally with the @if directive, such as displaying game over messages or player hands.
List Rendering:
Traverse list data like player hands or discard piles with the @foreach directive, dynamically generating corresponding UI elements.
CSS and Styling:
Define component styles with <style> blocks, setting layout, color, font, and more.
Apply styles to specific UI elements with class and style attributes.
JavaScript Interoperability:
Inject the JavaScript runtime with @inject IJSRuntime.
Call JavaScript functions with InvokeAsync() to implement advanced UI effects like animations and sounds.