Welcome!

Welcome to Tatedrez! Let this homepage be the index to this documentation.But first let me explain the main objectives I had in mind when developing this technical test.

Following SOLID principles.

Making sure to follow good practices will allow us to create more understandable, flexible and maintainable code. These principles will make Tatedrez more robust, scalable and easy to refactor if the project keeps growing.

Flexible game design

Flexibility is crucial when designing a game. I wanted to make sure the project could take various gameplay scenarios and future enhancements so the approaching was:

- Piece Customization: The game is built with the idea that new pieces or modifications could occur. From simple color customization to whole new pieces.
- Board configuration: A dynamic board, not only in aspect but in size and shape.
- Rule modifications: The rules are there to be broken… or at least modified. Some set of rules can be changed to allow pieces to jump over other pieces or having 5 pieces per player or need more than 3 consecutive pieces to win.

Simple but appealing

A puzzle game needs to be straight to the point, no visual noise. I tried to have a simple art style but with appealing movement and animations to have the player engaged and give some personality to the game.

Architecture explanation

Tatedrez uses an MVC architecture. It's simple but effective for our needs. For me, it was critical to decouple logic and maintain single responsibilities. I also find this pattern easy to maintain and scale. Additionally, it's flexible enough to implement my ideas.

In this section, I'll explain the main scripts. You may find other scripts not discussed here due to their importance or size.

Model:

- **GameDataSO**: ScriptableObject with the core data we need, both visually (pieces materials and board materials) and rules related like board size, consecutive pieces needed to win etc…
- **ResultScreenCatalog:** Small set of data, purely visual to represent the outcome of the game.

Views

- **APieceView:** Abstract class that will use to implement the playerPieces and the AI pieces
    - PlayerPieceView: In charge of receiving the player input.
    - AIPieceView: In charge of representing the IA movements.
- **UIView**: Main UI view, we can find other minor views that work with that one like the *HUDView* or the *GameFinishedPanelView*. I prefered to divide the views to have clear separate responsibilities.
- **TileView**: Small view to animate tiles and showing other FX.

Controllers

- **GameController:** Responsible for the game's logic. This includes controlling turns, ending the game, and managing dead-end scenarios (situations where no one can win due to piece placement).
- **AIController:** This controller is responsible for placing and moving the AI pieces, but the logic is implemented elsewhere. I chose to create different AI difficulty levels that can be set in the GameDataSO. To achieve this, I created the following classes:
    - **AAILogic:** This abstract class implements the different AI difficulties. It inherits from the ScriptableObject class. This way, we can easily assign different implementations to the GameData. I chose this system to enable future game difficulty progression by reassigning the different implementations as needed.
        - **AIEasy:** This class allows for completely random placement and movements.
        - **AIMedium:** This class uses the minimax algorithm for placement and movements, allowing us to assign values to different movements. We

can also configure how “human” we want the AI to be and allow for occasional mistakes.

- **TileController**: A small controller designed to manage tile states.

Other Important Classes

- **BoardManager**: This could definitely be considered a Controller. My main reason to call it manager was to make it clear that this class will be not only controlling the board logic but instantiating and managing other controllers. The BoardManager is in charge of:

    - Instantiating different classes such as the TileControllers and TileViews, the piece views and the GameController and initializing everything with the proper parameters.
    - Setting up the board size and shape reading the parameters from the GameData, Taking care of calling the different views to trigger the animations.
    - Takes care of all the logic related to the tiles like Highlighting them, validating a move etc
    - Takes care of some Player and AI logic, like checking which tiles are available to move into and actually calling the PieceView to reflect the changes.

  It's important to note that the movement logic of the pieces is not managed by a controller, but by the board manager. This might be considered a violation of the single responsibility principle.

  I found that a piece controller was nearly redundant. The logic for different movements is already implemented in an abstract class which I'll explain next. The BoardManager calls this abstract class and receives the list of tiles for each specific movement behavior. Therefore, a PieceController was just an unnecessary step, making a call to that logic which then had to be passed back to the BoardManager.

- **AMovementBehaviour:** This abstract class is used to implement different movement behaviors. It takes the current piece tile, the matrix size, and a list of free tiles as inputs. Different implementations then return a list of coordinates that the chess piece can move to. This system allows us to create as many pieces as we want and expand their behaviors. Access to this abstract class is through a ScriptableObject *MovementTypeSo*, which also enables us to create different versions and freely assign them to the pieces. Currently, we can only specify if the same behavior takes into account whether another piece is in the middle.

    - **BishopMovementBehaviour, KightMovementBehaviour, RookMovementBehaviour…**

Utility Classes

During the development of Tatedrez, I used various classes outside of the MVC, including an event dispatcher. Initially, I considered the singleton pattern, but its susceptibility to errors and scalability issues led me to adopt a different approach. This allowed me to access various utility classes throughout the code.

- **ServiceLocator:** This class serves as our Singleton. It is used to create and store different utility classes (referred to as services) employed throughout the code. The instantiation is executed in a specific place to prevent duplicates and other complications.
    - **GameConfigService:** This class provides access to GameData, ensuring it is not modified during runtime.
    - **EventDispatcher:** This is an event dispatcher.
    - **CoroutineRunner:** This service allows us to run coroutines from any class.
- **PersistantSceneLogic:** This class, located in a different scene, is where we create our ServiceLocator Singleton and register all its services. This process simplifies Singleton instantiation, reducing potential issues.

This pattern has the added benefit of allowing the same services to be instantiated multiple times in specific situations. For instance, we may need two EventDispatchers, one for Analytics events and another for Gameplay events.

Other Decisions And Considerations

There are other decisions not related with the architecture directly that I took in mind when developing the game.

Animations and Effects.

Due to performance issues with Unity Animation Controllers, I chose not to use them. For the animations we need, we don't require any complex features from the animator controller. I decided to completely abandon Unity animations and implement our simple animations using tweeners. This approach is more efficient overall (even if the impact is minimal), particularly in giving me more control over timings.

URP vs Built-in Render Pipeline

I'm a big fan of the URP for its ability to customize lighting, shadows, post-processing, and other elements. However, for this project, I chose the Built-in Pipeline due to time constraints and my own limitations. As I'm not an artist, I don't have the assets necessary to fully utilize the URP. To avoid compatibility issues with free assets from the store, I decided this was the best approach. Given more time, I would definitely switch to URP upon project completion.