Blocks Puzzle Game Report

Design

The Blocks Puzzle Game was implemented using a modular design approach, following the Model–View–Controller (MVC)** architecture to separate responsibilities and ensure maintainability:

Model: The `Model2dArray` class encapsulates the game's core logic, such as grid management, shape placement, and scoring. This class is designed to be independent of any user interface, ensuring clear separation of logic.
View: The `GameView` class is responsible for the graphical representation of the game. It handles drawing the grid, shapes, ghost pieces, and highlighting poppable regions.
Controller: The `Controller` class processes user input and coordinates updates between the model and the view.

Design Patterns Utilized:
Factory Pattern: The `Palette` class follows a factory–like behavior by generating new shapes and sprites. This abstraction allows for easy extension if additional types of shapes need to be introduced.
Observer Pattern: While not explicitly implemented, the `Controller` class observes changes and updates the view accordingly, resembling an observer behavior.

Additions/Extras:
Ghost Shapes and Poppable Regions: Ghost shapes are displayed when dragging, showing potential placement. Poppable regions are highlighted, enhancing player interaction.
Dynamic Palette Replenishment: After placing a piece, the palette replenishes automatically, keeping the gameplay smooth and engaging.

Software Metrics

The software metrics analyzed during the development include:

Cohesion and Coupling:
High Cohesion: Each class has a distinct role. For instance, `Model2dArray` only handles game logic, while `GameView` is exclusively responsible for UI.
Low Coupling: The interaction between `Controller`, `Model2dArray`, and `GameView` maintains low coupling, promoting better maintainability.

Cyclomatic Complexity:

Critical methods, such as `canPlace` and `isGameOver`, were reviewed for cyclomatic complexity. Where needed, refactoring was applied to ensure that complexity remained manageable, improving code readability and testability.

Code Coverage:
Unit tests were written to ensure that all methods, especially in `Model2dArray`, were well-covered, with a focus on the game logic, including shape placement and score updates.

UML Class Diagram

A UML class diagram has been created to illustrate the relationships between key classes (`Controller`, `GameView`, `Model2dArray`, and `Palette`). The diagram highlights the dependencies and how each component interacts to support the MVC structure.

Controller interacts with both `Model2dArray` and `GameView`.
GameView is dependent on `Model2dArray` for retrieving game state and `Palette` for the shape information.
Palette provides new `Sprite` objects for use in the game, creating a dependency from `Controller` and `GameView`.

Game
The game features a 9x9 grid where players place pieces to fill lines and clear regions. Key elements include:
Shape Placement: Players can drag and drop shapes from the palette to the grid. Ghost shapes help visualize where a piece can be placed.
Score System: Points are awarded for placing shapes and clearing regions.
Missing Features: The game lacks a tutorial or guided play mode, which could help onboard new players more smoothly. Additionally, animations for piece placement and region clearing could improve visual feedback.

Conclusion
The Blocks Puzzle Game successfully implements a modular MVC design, resulting in clear separation of concerns and ease of maintenance. However, there are areas for improvement:
Enhance the Observer Pattern: Explicitly implement the observer pattern to improve responsiveness between the model and view.
Improve Code Reusability: Refactor certain UI rendering methods in `GameView` to make them more reusable and testable.

Additional Tests
– Stress tests were added to ensure proper behavior when placing multiple pieces consecutively.
– Edge cases, such as attempting to place shapes on occupied cells or near the borders, were thoroughly tested.

Reflections
Java's strong object-oriented features and type safety made it well-suited for this project, ensuring a clear separation of concerns and robust error handling. However, compared to other languages like Python, the verbosity of Java can make rapid prototyping slower. Kotlin, with its concise syntax and Java interoperability, could be an interesting alternative for future development.