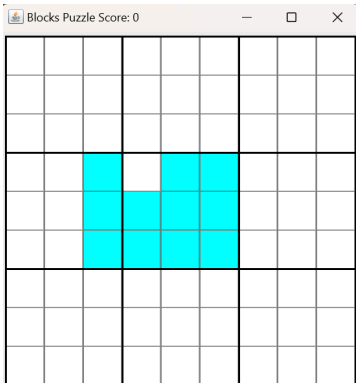
Block Puzzle Game Report

Game

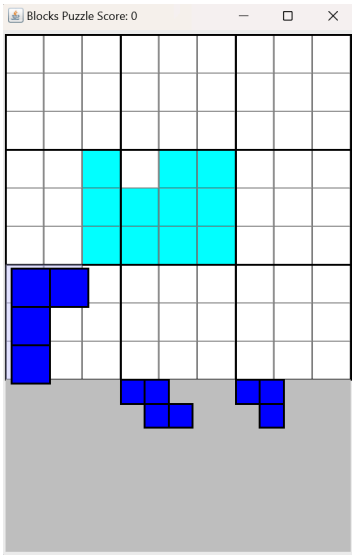
The key features of the game involve a 9x9 grid of cells that are initially unoccupied but can be occupied by dragging shapes over any section. Successfully completing a region results in its cells becoming unoccupied and the player gaining points.

Key features:

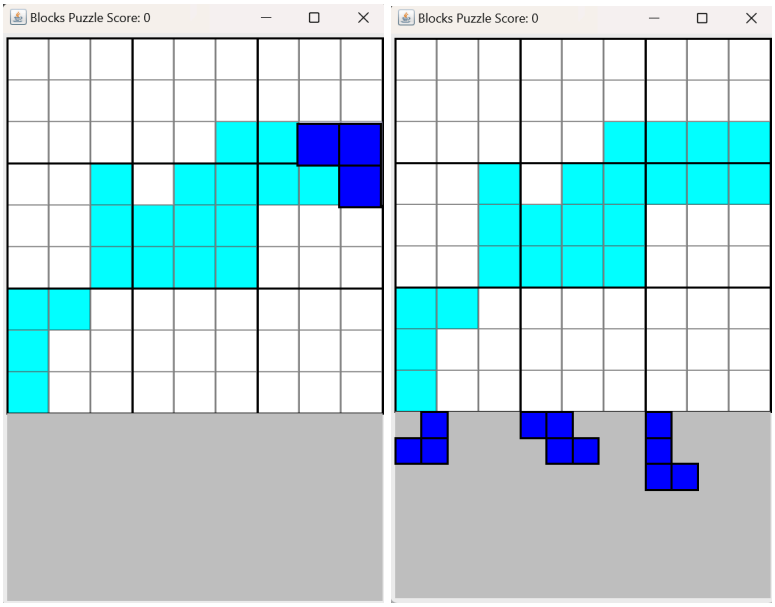
Full/empty grid cells



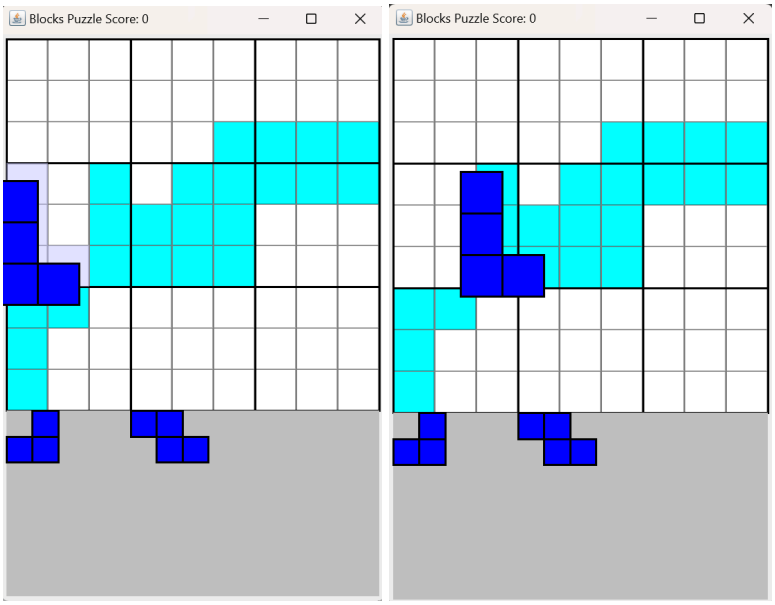
Drag and droppable sprites as pieces



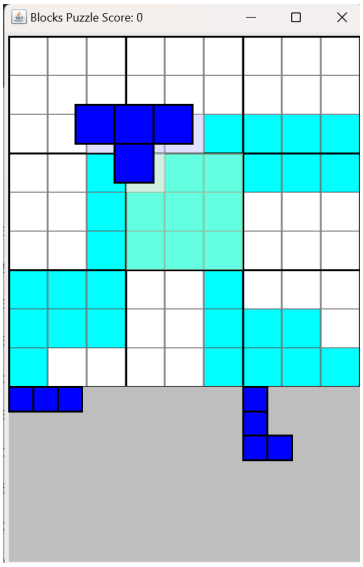
Palette display and replenishment



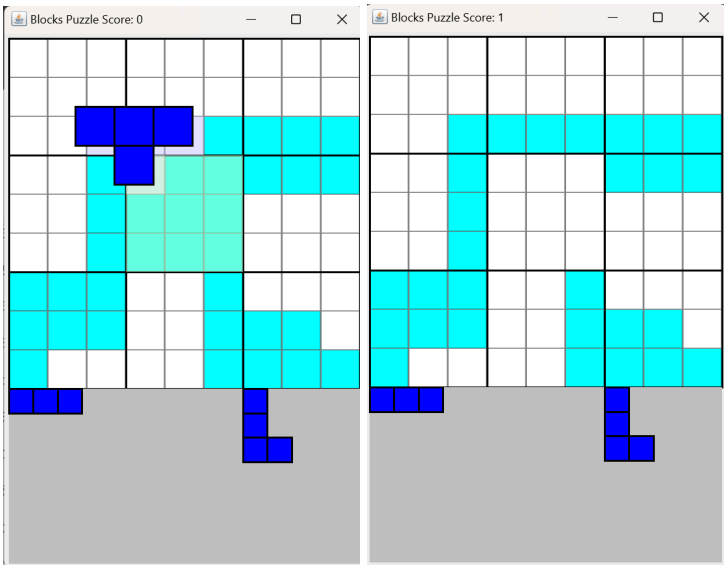
Ghost pieces for legal placements



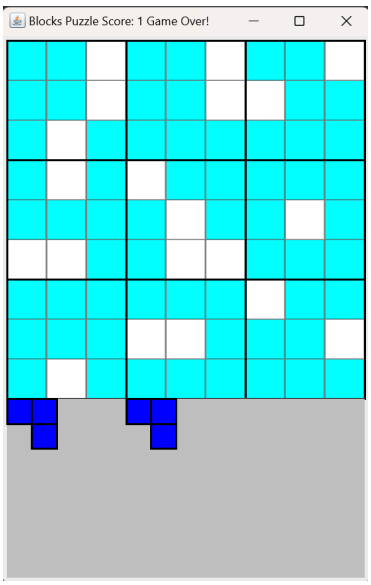
Poppable highlighting



Scoring

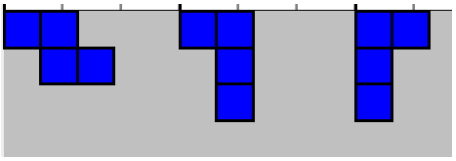


Game over detection

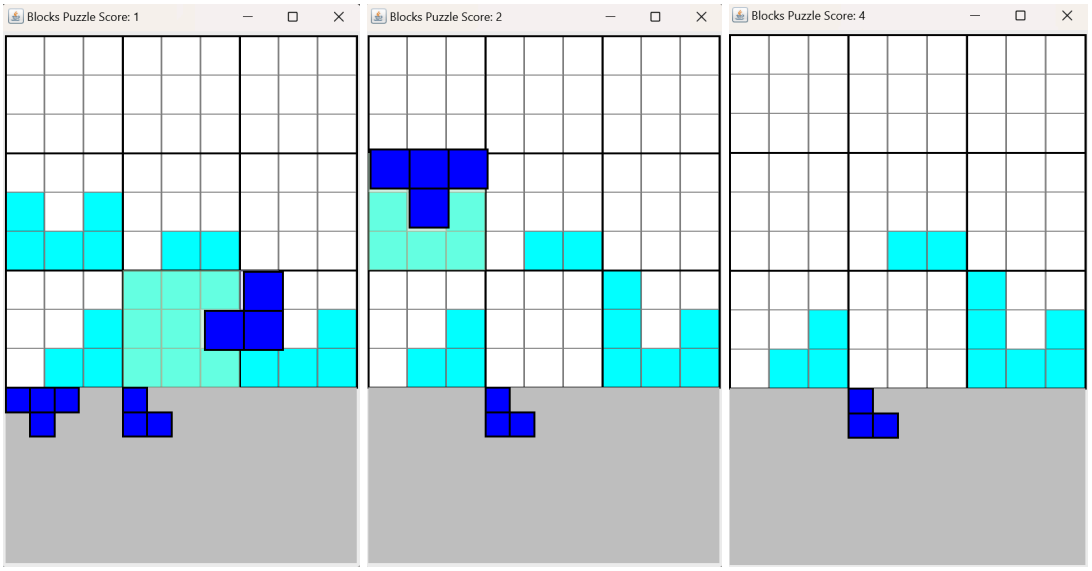


Improved shape set

```
public class BlockShapes {  new *                                                    3
    ic static class ShapeSet {  4 usages  new *
    // a good-enough set of 'Blockonimo' shapes
    ArrayList<Shape> shapeTypes = new ArrayList<>(List.of(  1 usage
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 2,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 0,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 0,  y: 1), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 1), new Cell( x: 1,  y: 0), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 1,  y_
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 2,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 0,  y: 1), new Cell( x: 0,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 0,  y: 1), new Cell( x: 0,  y:
            // Additional shapes
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 0,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 0,  y: 1), new Cell( x: 1,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 1,  y: 0), new Cell( x: 0,  y:
            new Shape(List.of(new Cell( x: 0,  y: 0), new Cell( x: 0,  y: 1), new Cell( x: 0,  y:
    ));
```



Score streak



Design

The main pattern that the Block Puzzle game I have created utilises is a Model-View-Controller pattern through the Model2dArray, GameView and Controller classes: Model2dArray implements the ModelInterface interface to handle several aspects of the game logic, such as whether a piece can be placed on the grid, identifying poppable regions and removing completed regions; GameView displays the current state of the game, including the grid, available sprites and ghost shapes; and Controller manages the user's interactions, such as clicking, dragging and releasing the mouse. The separation of these components allows for each to have specific responsibilities by separating their concerns, resulting in easier maintenance and testing.

The program also uses an observer pattern to update GameView through the Controller class. This is done using *view.repaint()* to update the view whenever the model changes. For example, if a sprite is placed in the grid, the Controller class calls *view.repaint()* to repaint the grid with the updated information.

Additionally, the program utilises a strategy pattern through the Shape object, which allows for easy accommodation of new types of shapes without amending methods such as *canPlace*.

Software Metrics

Using the provided metrics file, the software metrics for the program showed the following

Lines of Code

blocks.BlockShapes - 66

blocks.Controller - 84

blocks.GameView - 108

blocks.Model2dArray - 115

blocks.ModelInterface - 13

blocks.ModelSet - 59

blocks.Palette - 85

blocks.RegionHelper - 67

blocks.State2dArray - 6

blocks.StateSet - 4

The relatively low lines of code across the files reflects the overall simplicity of the program, resulting in the code being easier to read and maintain. Some of the files, such as Model2dArray and GameView are larger than others such as BlockShapes and RegionHelper, which may indicate higher complexities in the former.

Cohesion

LCOM refers to the cohesion of a class, and counts the number of unconnected methods that do not share the same instance or static variables, or if they do not call each other. The LCOM for each file of the program is:

blocks.BlockShapes - 3

blocks.Controller - 0

blocks.GameView - 0

blocks.Model2dArray - 53

blocks.ModelInterface - 28

blocks.ModelSet - 49

blocks.Palette - 0

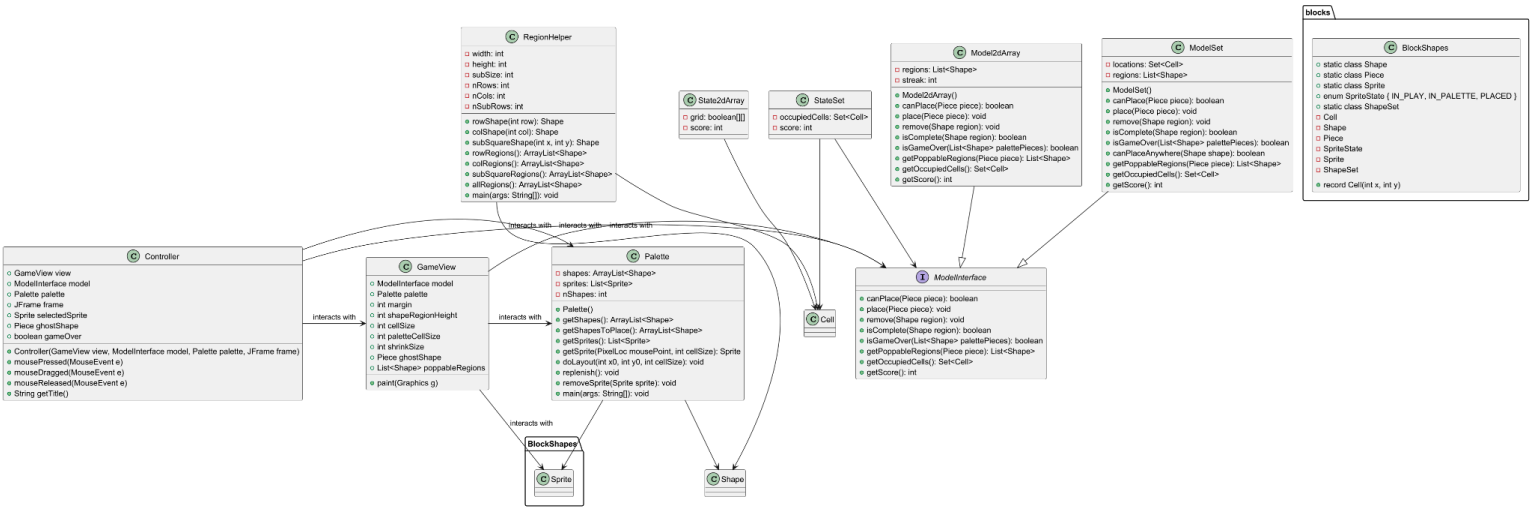
blocks.RegionHelper - 28

blocks.State2dArray - 0

blocks.StateSet - 0

The low LCOM of some files, such as Controller, GameView and Palette indicate high cohesion, meaning that each would be easier to test and changes to one method are less likely to affect the rest of the class or have unintended side effects. On the other hand, files such as Model2dArray, ModelSet and RegionHelper have a high LCOM, especially Model2dArray, which indicates that breaking the file down into smaller, more focused classes could be beneficial. However, due to the variety of tasks that the class performs, a higher LCOM may be justified compared to the rest of the classes.

UML Class Diagram



Conclusion

In conclusion, the Block Puzzle game has been implemented using design patterns such as a Model-View-Controller, Observer and Strategy pattern. The separation of the Model (Model2dArray), View (GameView) and Controller (Controller) allows for easier maintenance of the code. The addition of an extra test, *testScoreBoost*, ensures the additional, more rewarding scoring system functions as intended. However, some aspects of the code could be improved, such as the cohesion of some of the classes such as Model2dArray and RegionHelper. Additionally, the random play mode feature, which involves watching a bot play the game, could have been implemented given more time spent developing the game.

The choice to use Java for this project was beneficial due to Java's strong support for object-oriented programming, which allowed the program's heavy reliance on objects and classes (such as Piece, Shape and Cell) to easily be designed and implemented. The use of interfaces and inheritance allows for more modular game logic and use of either Model2dArray or ModelSet, which both implement ModelInterface. A disadvantage of using Java over a language such as C++ is its generally slower performance, however due to the relative simplicity of the program as a whole, this slightly slower performance is not a major issue.