Lewis Day
220318228

Block Puzzle Lab Assignment Report

Design

The whole application builds on the model-view-controller design pattern (MVC) as there are specific classes for displaying the view, creating the model and the controller. The model is the interface class ModelInterface, with the implementations being the classes Model2dArray and ModelSet. These provide the main logic of how the pieces and shapes can be placed and checks whether the game is over. The view is class GameView, which manages the painting of several aspects of the game for example, the palette, grid, ghost shapes and poppable regions. The controller class called Controller manages all aspects of the game by creating instances of the other classes and uses mouse listeners to manage what logic should take place in the selected model and the GameView classes. MVC is effective as it creates a separation of concerns, allowing for better maintainability and testability.

Another design pattern that is used is the observer pattern, where the mouse listeners in the Controller class are the observers. These observe for an action to occur, a mouse click, drag or release.

Software Metrics

In terms of lines of code metric (LOC), class GameView has a high LOC number at 124 lines of code however, there is a lot of functionality in this class, which means that this isn't necessarily a negative. The next class I will look at for LOC is Model2dArray, which has an LOC count of 114 lines, this is also due to the amount of functionality which takes place in this class. In contrast to Model2dArray, ModelSet which is the alternative implementation of Model2dArray, has a LOC count of 67. This is due to the fact that many of the functions are done using streams, which are usually only a 1 to 3 lines instead of the multiple lines used in Model2dArray to do iteration. Palette is another class with a reasonable LOC metric at 76, even though this class has quite a few functions to carry out, LOC is still relatively low. Finally, Controller has slightly higher LOC than Palette, but this is due to the fact that more logic takes place here. Regarding the LOC for the classes above, LOC is not a determining metric, and other metrics should be used as well to make a decision.

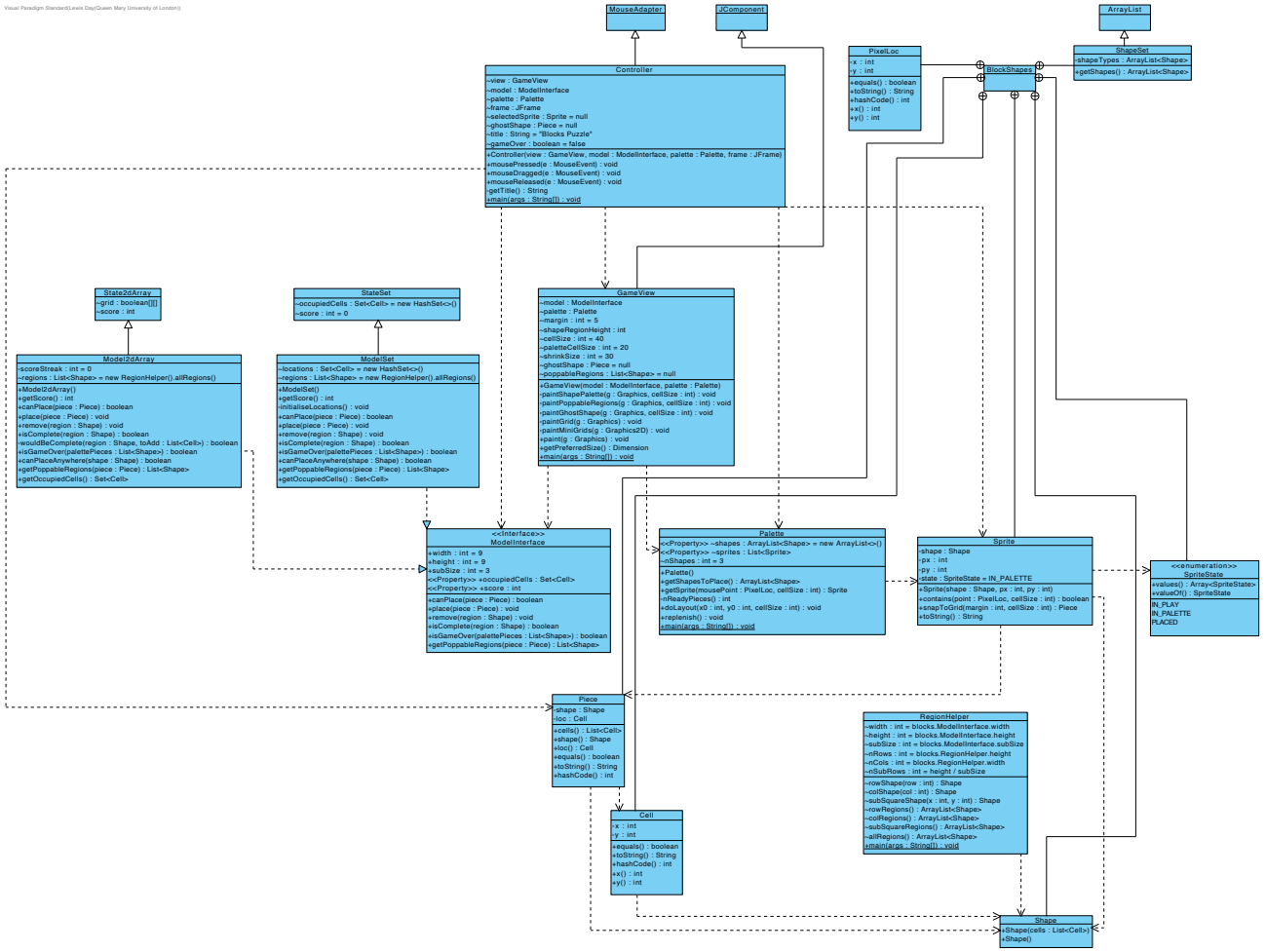
The next metric which is a greater determinant of how well code has been written is the cyclomatic complexity. Class Palette has a complexity of 14, with most methods having a complexity of 1, with the following having a cyclomatic complexity of 2: getShapesToPlace, nReadyPieces and getSprite. There was one method in Palette which has a cyclomatic complexity of 3, which is replenish. The next class is Model2dArray, which has a class cyclomatic complexity of 27. This is quite high for a class however, it will be due to the implementation of this class as it requires lots of iteration through lists and conditionals for each method, which are contributors to cyclomatic complexity. Most methods in this class have a cyclomatic complexity of 2 or 3. Two methods in this

class have the highest cyclomatic complexity out of any methods at 4, which are getOccupiedCells and canPlaceAnywhere. Despite this, there are two methods with a value of 1, which are remove and isGameOver. Controller has a cyclomatic complexity of 12, with most methods being 1 or 2 however, 2 methods mouseReleased and mouseDragged have a value of 3 each. ModelSet has a much lower cyclomatic complexity compared to Model2dArray at 16, this is due to its use of streams. Majority of the methods in this class are 1 or 2 (primarily 1) and 1 method has a value of 3 which is initialiseLocations. The final class is GameView, with a class value of 16. Generally, most methods in this class have a value of 1 or 2 with two methods with a value of 3 being the following: paintMiniGrids and paintGrid.

Coupling between objects (CBO) is 8 for GameView, Controller and Model2dArray, 7 for ModelSet and 5 for Palette. GameView and Controller have high coupling between objects because they depend on other objects and classes in order for the game to function. This is understandable; however, it could be reduced.

The final metric I will analyse is lack of cohesion in methods (LCOM). Palette, Controller and GameView each have an LCOM of 0, which means there is high cohesion in the classes. However, Model2dArray and ModelSet have much higher LCOM metrics at 55 and 53 respectively, which means that there is low cohesion in the classes, and this isn't ideal. As a result, this indicates that these classes could do with splitting up and refactoring in order to improve this metric.

UML Class Diagram

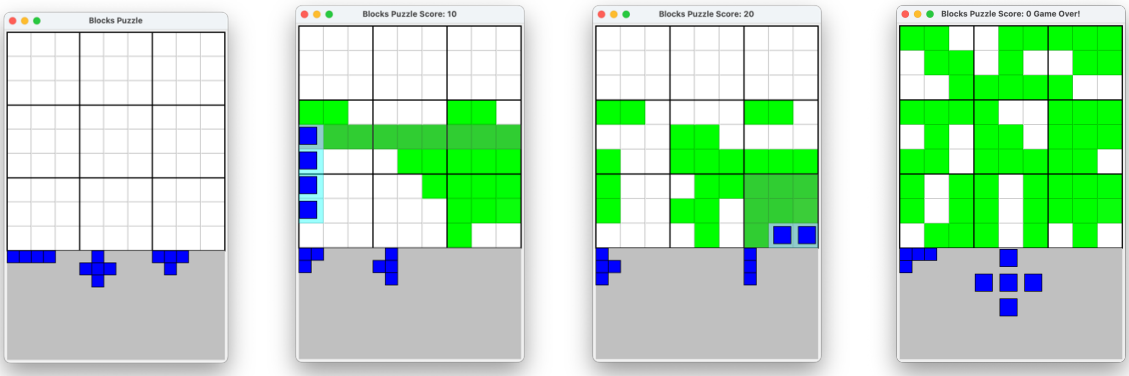


Game

My implementation supports all standard features of the game. The palette holds all of the available shapes which can be placed on the grid, which is generated at random. Each shape in the palette can be selected, dragged onto the board and dropped at a particular location. When a shape is dragged across the board, and is over a legal placement of that piece, a ghost shape appears in cyan, which means that the shape can be dropped at that location. If a shape is dragged over an occupied location, or somewhere that it cannot be placed, the ghost shape doesn't appear. Placing a piece on a legal location, where the ghost shape appeared, the cells will turn green, showing that this location is now occupied. As the game progresses, there may come a time where a row, column or a 3x3 square becomes completely occupied when a shape is placed, a 'poppable' region is created. On creation of a 'poppable' region, the region is highlighted and when 'popped' the cells become unoccupied again. This links in with the scoring system of the game, where a standard pop is 10 points and there is a score streak system for 'poppable' regions, where you get more points for the more successive 'poppable' regions complete. The score streak operates as a score multiplier, this is an extra feature. The other extra feature I implemented was more shapes which could populate the palette. A game will end when the game checks the palette, and a piece cannot be placed, or you try to select a piece that cannot be placed on the grid. The only extra feature that isn't available is the random play mode, where a bot plays the game.

Screenshots

These screenshots show various different states of game play.



Conclusion

Overall, the metrics are generally good for this assignment. However, there are some classes which could be refactored to improve the metrics. Furthermore, this assignment could be improved by implementing a bot which allows for a random play mode for the game. Also, another design pattern could be added like the factory pattern to manage the creation of shapes or the strategy pattern could be used to decide between Model2dArray and ModelSet. In terms of other languages, Python would not be suitable for this due to its lacking graphics capabilities compared to Java.