FOOP Report on Block Puzzle

Design

The program makes use of a Model View Controller design pattern which works great because it provides a nice separation of concerns and makes coding a lot easier. It also makes use of the strategy design pattern for the model which is core as it allows us to test our code with these two different models and evaluate them. The program as a whole had a lot of nice encapsulation and modularity in its design which made it really easy to find where an error was coming from. One implementation that I did was the additional task of Random play which I decided to include within the Controller class, although in hindsight it might've been better to create a separate class for this

Software Metrics

Classes	WMC - Before	WMC - After
Model2dArray	13	35
ModelSet	13	18
Palette	13	19
GameView	14	22
Controller	8	31

Looking into the metrics of before and after the finished program, we will see that some metrics such as WMC would have increased throughout most of the classes which makes sense because additional complexity is required for the functionality.
In terms of the two different models, I have found that ModelSet generally required much less complexity to get the functions serving their purpose and this seems mostly due to the use of functional programming such as streams. A lot of functions that served the same purpose in Model2dArray required more lines of code and additional complexity whereas ModelSet functions, a lot of them were completed in a single line of code at least for my case.

Complexity increases within Palette, GameView and controller are not surprising especially when additional work has been included. Controller specifically, holds a method for random play which brings additional complexity along with it.

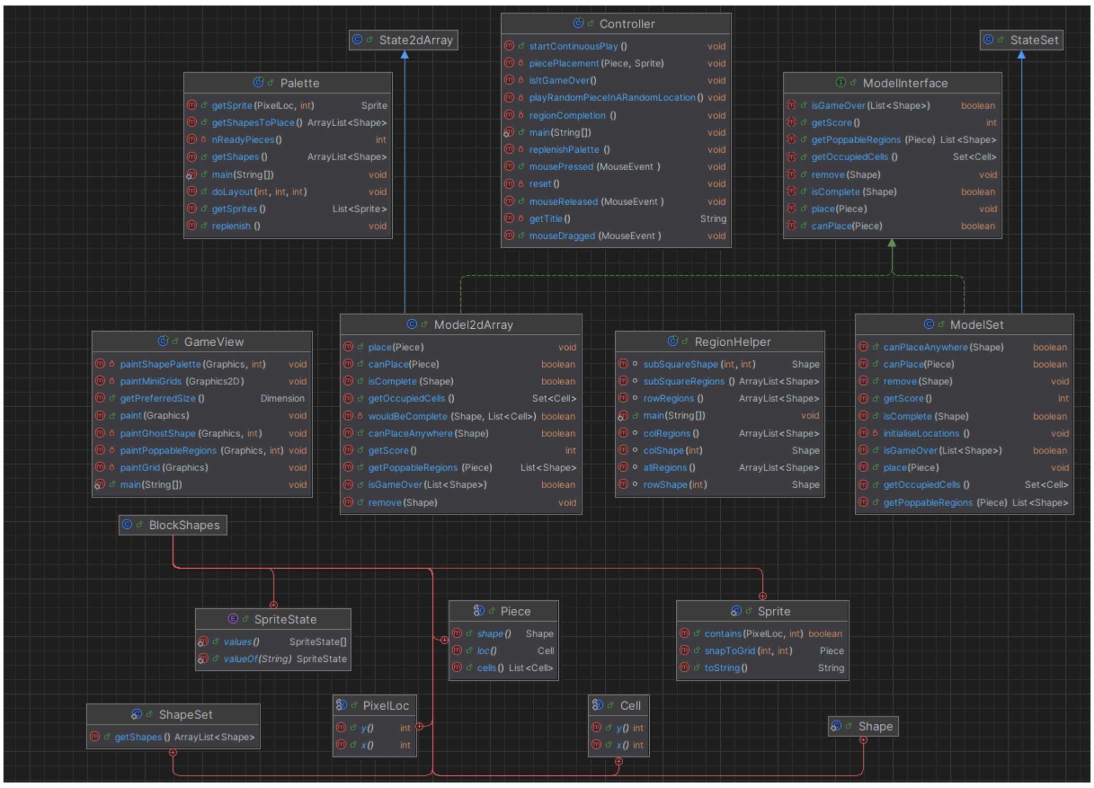
Classes	CBO - Before	CBO - After
Model2dArray	7	8
ModelSet	7	7
Palette	5	5
GameView	8	8
Controller	8	10

CBO of classes generally stayed very similar with some exceptions in Controller and Model2dArray. Model2dArray seemed to increase in CBO after iterating through regions which is a needed operation. Controller has an increase in CBO due to additional work such as needing to import a timer for Random Play functionality.

Complexity and Coupling has stayed very similar throughout the program after completion, however it seems that the Set model works a bit better than its 2d Array counterpart in these areas with no additional coupling introduced and way less complexity included. Although this might be due to my own implementations, it must be said that the point still stands as working with the set model seemed far easier and less work than the 2d Array.

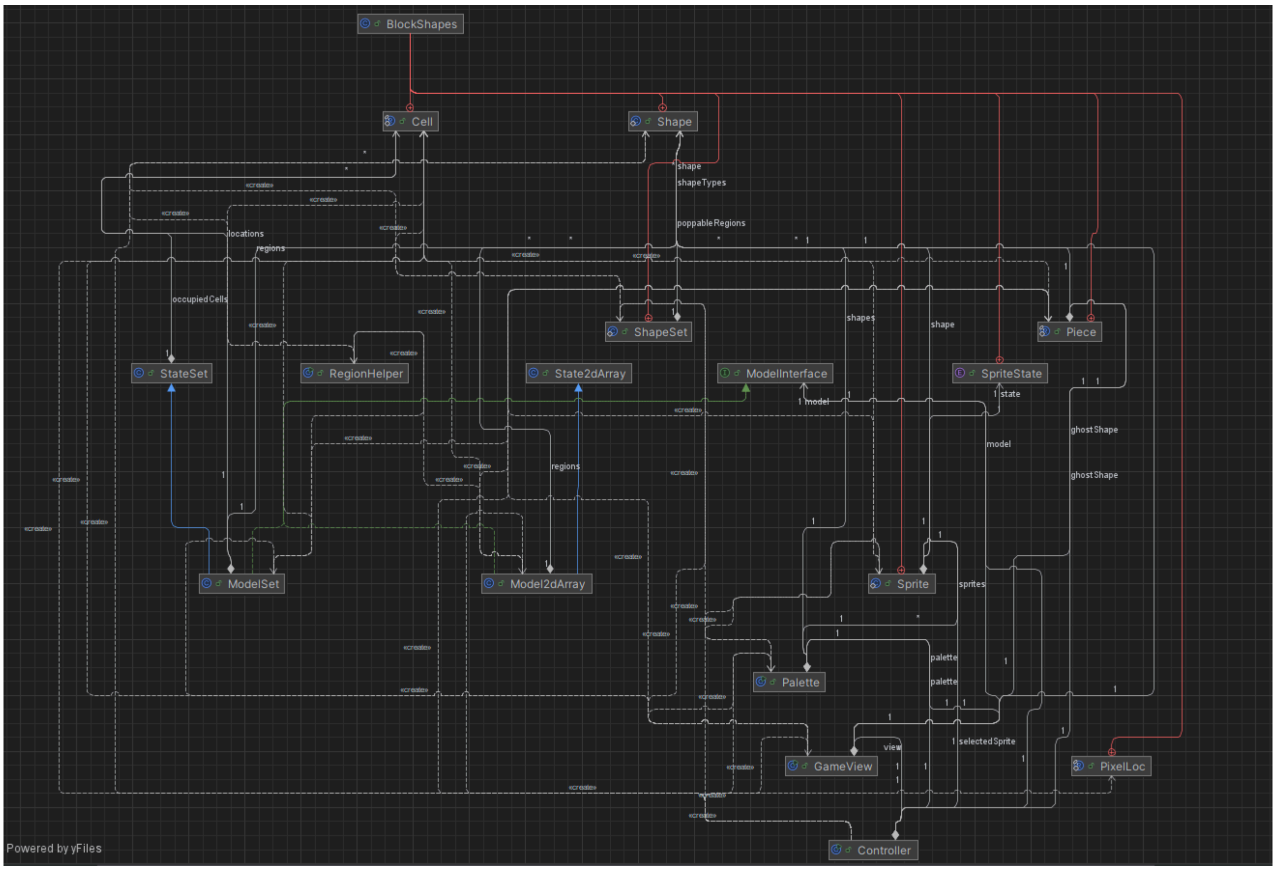
UML Class Diagram

This is the UML diagram with Inner classes and methods included. This does not show dependencies present within the program.

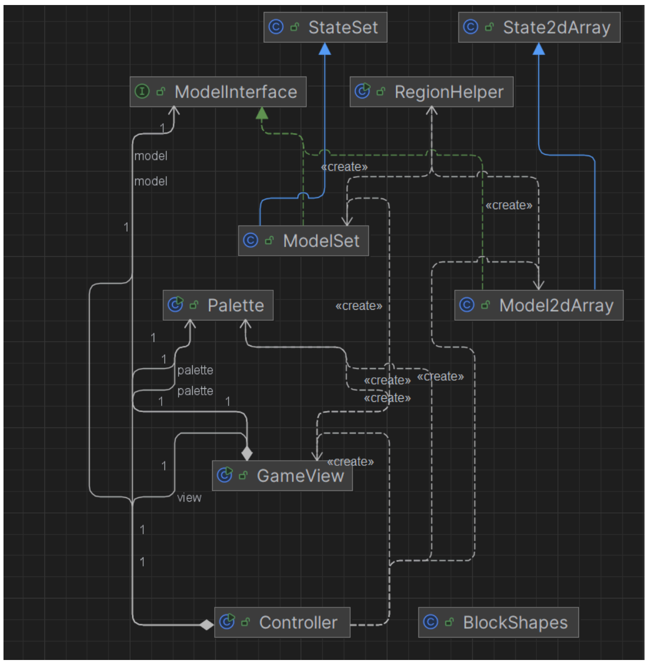


State2dArray
Controller
startContinuousPlay () void
piecePlacement (Piece, Sprite) void
isItGameOver() void
playRandomPieceInARandomLocation() void
regionCompletion () void
main(String[]) void
replenishPalette () void
mousePressed (MouseEvent) void
reset() void
mouseReleased (MouseEvent) void
getTitle() String
mouseDragged (MouseEvent) void
StateSet
Palette
getSprite (PixelLoc, int) Sprite
getShapesToPlace() ArrayList<Shape>
nReadyPieces() int
getShapes () ArrayList<Shape>
main(String[]) void
doLayout(int, int, int) void
getSprites () List<Sprite>
replenish () void
ModelInterface
isGameOver (List<Shape>) boolean
getScore() int
getPoppableRegions (Piece) List<Shape>
getOccupiedCells () Set<Cell>
remove (Shape) void
isComplete (Shape) boolean
place(Piece) void
canPlace(Piece) boolean
GameView
paintShapePalette (Graphics, int) void
paintMiniGrids (Graphics2D) void
getPreferredSize () Dimension
paint (Graphics) void
paintGhostShape (Graphics, int) void
paintPoppableRegions (Graphics, int) void
paintGrid (Graphics) void
main(String[]) void
Model2dArray
place(Piece) void
canPlace(Piece) boolean
isComplete (Shape) boolean
getOccupiedCells () Set<Cell>
wouldBeComplete (Shape, List<Cell>) boolean
canPlaceAnywhere (Shape) boolean
getScore() int
getPoppableRegions (Piece) List<Shape>
isGameOver (List<Shape>) boolean
remove (Shape) void
RegionHelper
subSquareShape (int, int) Shape
subSquareRegions () ArrayList<Shape>
rowRegions () ArrayList<Shape>
main(String[]) void
colRegions () ArrayList<Shape>
colShape (int) Shape
allRegions () ArrayList<Shape>
rowShape (int) Shape
ModelSet
canPlaceAnywhere (Shape) boolean
canPlace(Piece) boolean
remove (Shape) void
getScore() int
isComplete (Shape) boolean
initialiseLocations () void
isGameOver (List<Shape>) boolean
place(Piece) void
getOccupiedCells () Set<Cell>
getPoppableRegions (Piece) List<Shape>
BlockShapes
SpriteState
values() SpriteState[]
valueOf(String) SpriteState
Piece
shape() Shape
loc() Cell
cells() List<Cell>
Sprite
contains (PixelLoc, int) boolean
snapToGrid (int, int) Piece
toString() String
ShapeSet
getShapes () ArrayList<Shape>
PixelLoc
y() int
x() int
Cell
y() int
x() int
Shape

This is the UML diagram with Inner classes and dependencies included, however no methods. As you can see in the picture there are a lot of dependencies present in the program which if we look back to the CBO of different classes, this makes sense.

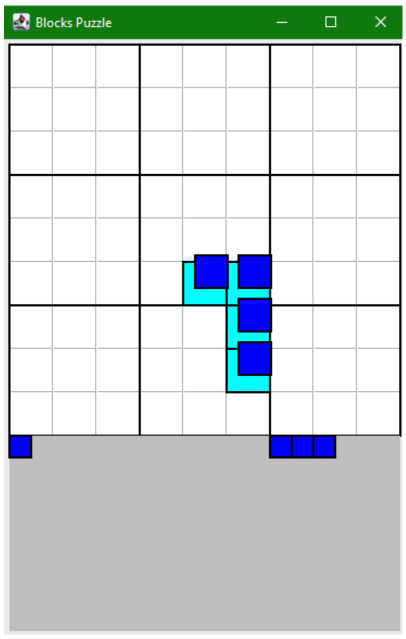


A much clearer image would be to exclude inner classes and this will allow us to view any dependencies between the main classes that we developed. However it is a missing a lot of information such as all the dependencies that exist with the classes inside of BlockShapes

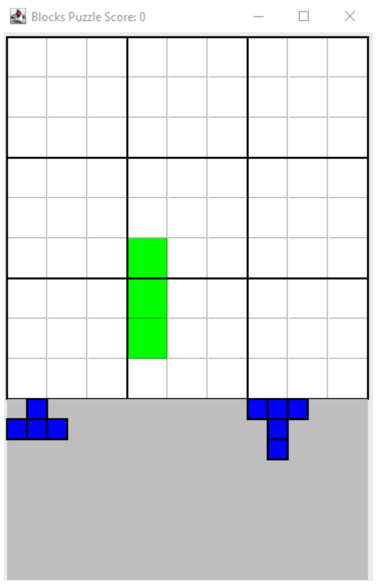


Game

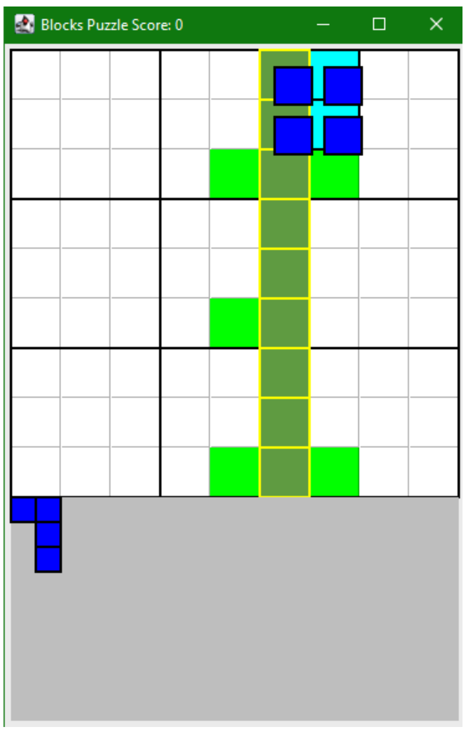
The game is about completing a row, column or a 3x3 grid of cells to earn points. The player can click on a piece visible at the bottom and drag it anywhere on the grid. It will then show a ghost piece behind it showing where it will be placed.

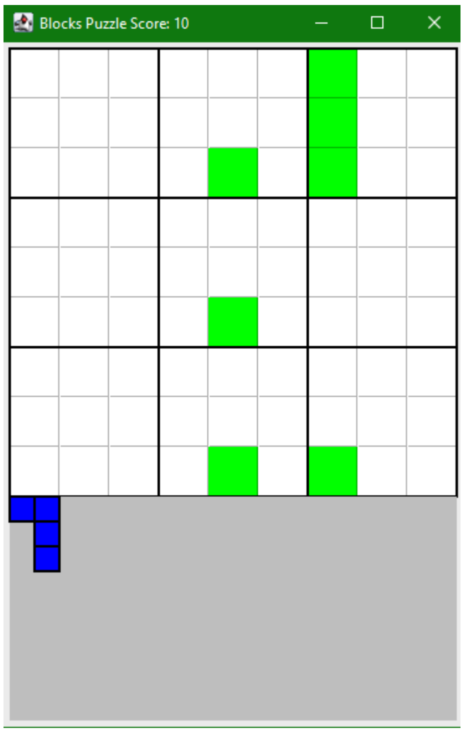


The player has a selection of shapes that they can place which is refilled only when the palette is empty. This is what the board looks like after a shape has been placed.

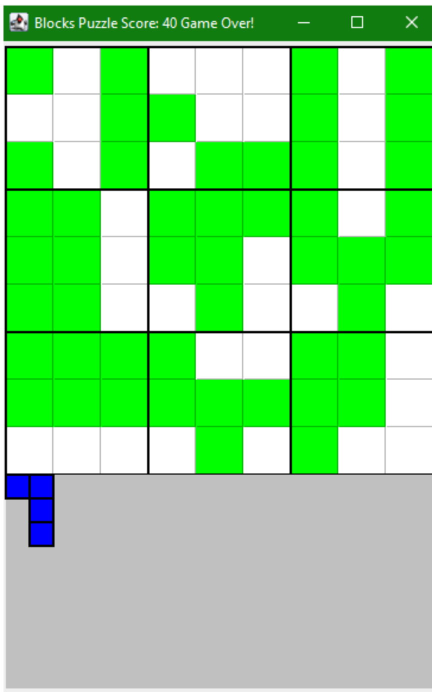


When the player completes a row, column, or 3x3, that space will be highlighted in green and yellow to show that it will be removed and that it will increment the score.

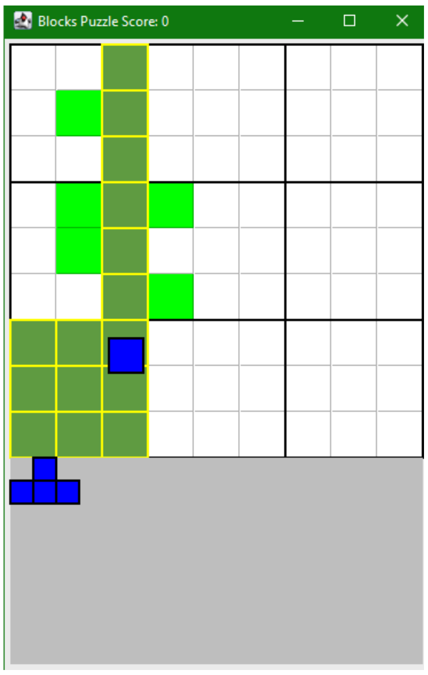


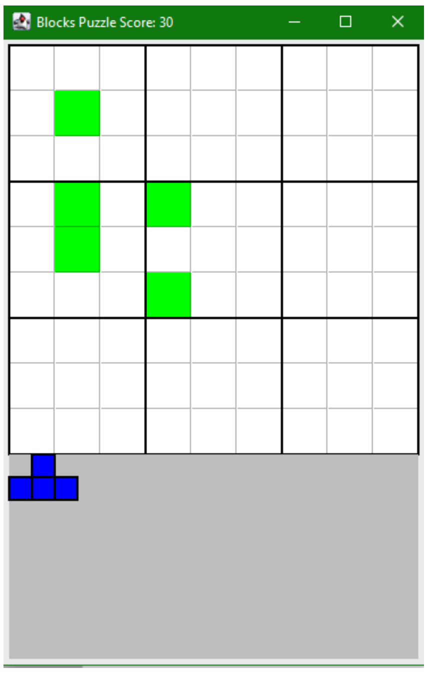


The player then continues to play until it is game over and they aim for the highest score. A game over is when the player can no longer place any of the shapes in their palette on the board. This is what a game over state of the game looks like.

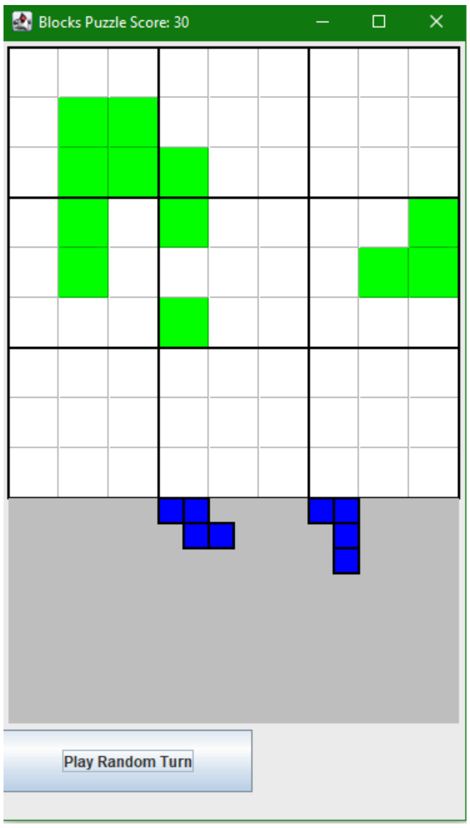


An additional feature that has been implemented is that the score will increase more for each subsequent completion that they get. Typically each completion gives you 10 points, however in the image below you can see that if i complete 2 at the same time or two in a row, my score is increased by 30 instead (10*1 + 10*2 = 30)





Another addition that I have done is extra shapes which can be seen over the different images. Lastly, I have added the automatic random play, which places random pieces in random locations which is great for testing.



Conclusion

The software design all in all works quite well for this purpose, the implementation of both models was definitely interesting but I believe that the ModelSet implementation is better for this purpose due to its lower complexity and overall code cleanliness. In terms of how Java compares to other languages, Java works quite well for this with Object Oriented Programming, however this kind of game might serve its purpose better on a website meaning languages such as Rust, C or C++ might be better because it can be compiled into Wasm. Languages such as Scala or Kotlin would also be really good alternatives due to their stronger support for functional programming which I did make use of in Java. Using Java,Kotlin and Scala also bring great portability between devices due to the Java Virtual Machine (JVM), which make them great choices for desktop games.