Report on Magic Squares

Overview

A magic square is an array of positive integers where each row, column and both main diagonals sum to the same total (Fig 1.1). Third order magic squares were known to Chinese mathematicians as early as 190 BCE[1].

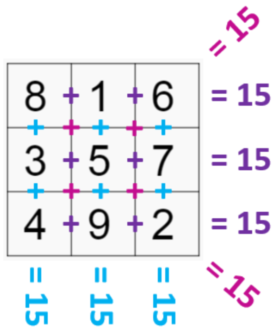


Fig1.1 – 3x3 magic square: each row, column and diagonals total 15[2]

From there, there are references of their usage in the sub continent before propagating westward and arriving in Latin Europe through the Middle East in the 14th century.

This report was asked to investigate a subset of magic squares, odd squares, and solve the following tasks:

- Write a command line application using Java™ that will generate a user desired size magic square using the pseudo code provided.
- Develop the magic square into a game which shuffles the elements and then allows the user to input moves to try and reconstruct the solved square.

The task emphasized using object orientated programming (OOP) principles and utilizing appropriate types, program control structures and classes from the core API (application programming interface).

Main Goals: The core features of object orientated design are:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

The code should adhere to the above so that the program is extendible, easy to maintain and DRY (Do not Repeat Yourself). However, adhering to another tenant of good coding practice, KISS (Keep It Simple Stupid), there is no point in utilizing structures and principles for the sake of it. As such, there is no overt need for inherited classes and a different approach will be utilized: **Composition**.

By using classes to compartmentalize the code and have one overlord class – GameManager – handle the main logic whilst delegating correlated tasks to other classes e.g. handling UI (user interface) and score logic, the advantages of inheritance can be replicated but using a more logical approach.

The solution will then be tested by manual checking, debugging using built-in IDE (integrated development environment) tools and running a battery of unit tests.

Solution Design

The Magic Square

Researching the algorithm revealed the provided pseudo code is actually a *reverse* of the Siamese algorithm, one devised in Siam and bought back by French mathematician De la Loubère[3] in the 17th century.

The algorithm starts by intiialzing 1 in the first row at its midpoint: x = 1, y = (n + 1) / 2 (where x and y correspond to grid positions i.e. one more than index positions). Then loop from 2 to n, where n is the grid size, inserting numbers in the square one to the left and up (i.e. top left diagonal) if free. If not free, insert directly below. Visually this is displayed in figure 2.1.

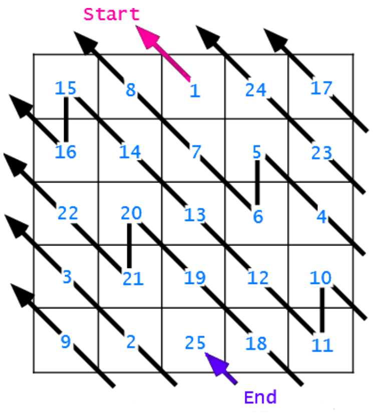


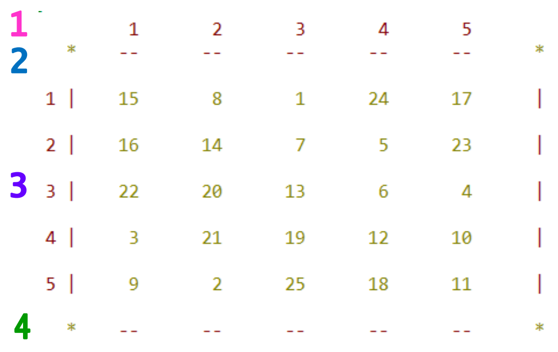
Fig2.1 – Shows the route taken by the algorithm in a 5x5 square

Programmatically, this was accomplished by creating an `initializeMagicSquare()` method which takes no parameters, stores x and y as local variables before following the pseudo-code verbatim.

Two modifications were needed to the pseudo code to get the desired output:

1. Two additional local variables were created, wrappedX and wrappedY to handle wrapping.
2. Before updating the grid, one needed to be subtracted from x and y to get the corresponding index position.

Printing of the solution was handled by an overridden `toString()` method. It utilized a StringBuilder to build up the grid (Fig 2.2) using four loops. Loop one was for column numbers (1). Loop two for top grid separators (2). Loop three for row numbers and values (3) – spacing controlled with tabs and maxStringWidth variable. Finally, loop four for bottom row separators (4). ANSI (American National Standards Institute) colour codes were added for aesthetics.



```
          1      2      3      4      5
    *     --     --     --     --     --     *

  1 |     15     8      1      24     17     |

  2 |     16     14     7      5      23     |

  3 |     22     20     13     6      4      |

  4 |     3      21     19     12     10     |

  5 |     9      2      25     18     11     |

    *     --     --     --     --     --     *
```

Fig 2.2 – Magic square generated with ANSI colours

For the game, a shuffling feature needed to be added to the MagicSquare class. The requirements asked for n^2 shuffles but only allowed swaps along a single axis. To accomplish this, the Random module was used to generate a number from 1 to 4. The result was fed to an enhanced switch case that allocated moves to each number: 1 = left, 2 = right, 3 = up, 4 = down (Fig 2.3).

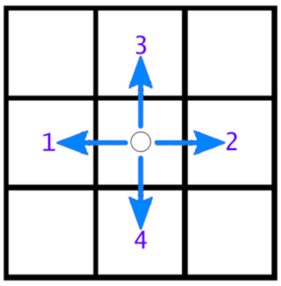
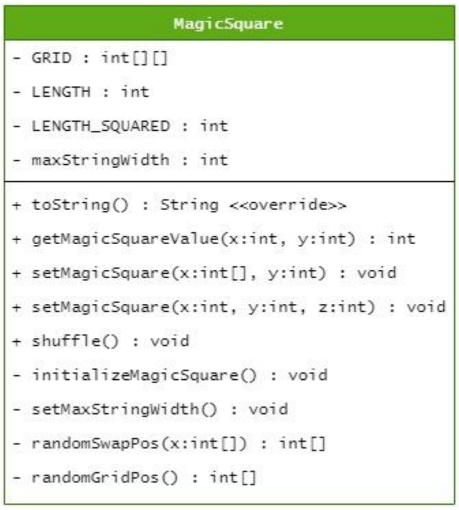


Fig 2.3 – Allowed swap positions for any given square

The shuffle method was a composite method that utilized three other methods to accomplish its goals:

- `randomGridPos()`: picks a random grid pos
- `randomSwapPos()`: picks swap square (Fig 2.3)
- `setMagicSquare()`: performs the actual swap

The final MagicSquare class can be represented as a UML (Unified Modeling Language) class (Fig 2.4).



```
MagicSquare
- GRID : int[][]
- LENGTH : int
- LENGTH_SQUARED : int
- maxStringWidth : int

+ toString() : String <<override>>
+ getMagicSquareValue(x:int, y:int) : int
+ setMagicSquare(x:int[], y:int) : void
+ setMagicSquare(x:int, y:int, z:int) : void
+ shuffle() : void
- initializeMagicSquare() : void
- setMaxStringWidth() : void
- randomSwapPos(x:int[]) : int[]
- randomGridPos() : int[]
```

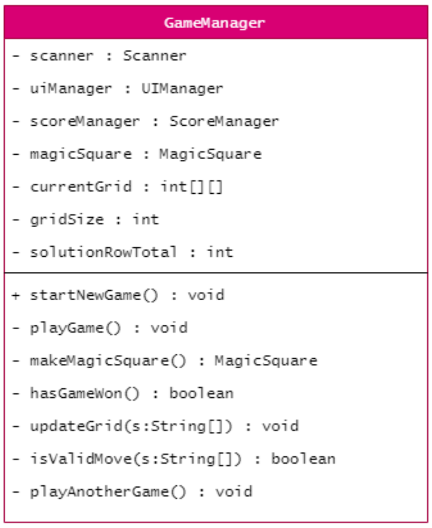
Fig 2.4 – UML Class Diagram of MagicSquare class

All the field variables are private following the encapsulation principle of OOP. For space reasons, the getters are not included in the method section in Fig 2.4 (but are in the code) – there are no setters as the class variables are final. The `toString()` method is an overridden method whilst `setMagicSquare()` is an overloaded method. They both exhibit the quality of polymorphism in Java™ where the same code can be used in different ways.

The Game

The game was designed using composition where there would be one overlord class that oversaw the main game logic but graphical elements and scoring logic was decomposed out into separate classes.

The GameManager class (Fig 2.5) was created to handle the bulk of the game logic including communicating with the MagicSquare class (aggregation) and instantiate UIManager and ScoreManager objects which controlled UI (user interface) and scoring logic respectively (composition).



```
GameManager
- scanner : Scanner
- uiManager : UIManager
- scoreManager : ScoreManager
- magicSquare : MagicSquare
- currentGrid : int[][]
- gridSize : int
- solutionRowTotal : int

+ startNewGame() : void
- playGame() : void
- makeMagicSquare() : MagicSquare
- hasGameWon() : boolean
- updateGrid(s:String[]) : void
- isValidMove(s:String[]) : boolean
- playAnotherGame() : void
```

Fig 2.5 – UML Class Diagram of GameManager class

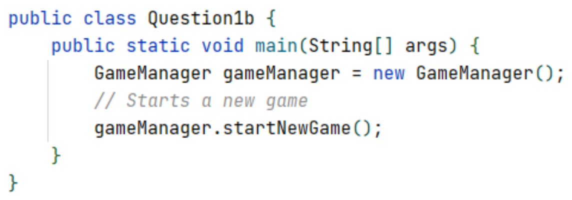
The GameManager class has two important methods to control game flow. The first is `hasGameWon()` which is run in a while loop and on each loop calculates the sum of every row, column and diagonal and compares it to solutionRowTotal. The second is the `isValidMove()` method which splits user input into three parts, ignores case and checks they match the desired input specs of:

Row Column Direction e.g. 2 1 U

Both of these methods are helper methods for the primary and only public method – `startNewGame()` – which wraps all the game logic into a single method demonstrating a high level of abstraction.

Composition

ScoreManager and UIManager have a range of public methods used by GameManager with UIManager leveraging a class with static fields, VFX, to produce some of the UI features. How all these pieces fit together is best illustrated in the full UML diagram (appendix 1.1). The key take home is that the final solution is called in a static main method in class Question1b which simply instantiates a GameManager object and then calls the `startNewGame()` method resulting in extremely clean OOP code.



```java
public class Question1b {
    public static void main(String[] args) {
        GameManager gameManager = new GameManager();
        // Starts a new game
        gameManager.startNewGame();
    }
}
```

Fig 2.6 – Question1b class which starts a game

Assumptions

- The 48-bit LCG (Linear Congruential Generator) algorithm used by the java.util.Random class is sufficiently random.
- The n^2 shuffle algorithm is enough to generate a sufficiently mixed grid
- The shuffle algorithm will not produce a solved board

Discussion

Solution Scope

The minimum features required were:

- Create an odd sized magic square from the user
- Shuffle the magic square
- A game where the user tries to reconstruct the magic square through command line inputs
- Output the number of moves to beat the game

All the above were implemented, of particular note my implementation not only prints out odd squares, it offers feedback for incorrect input as well as handles erroneous inputs.

The resultant print out works by overriding the existing `toString()` method illustrating code efficiency and showing the need to not create superfluous additional methods.

In addition, a number of added features were implemented.

- **High Score** – at the end of each play through, a hash map is updated with the user's lowest move total. The user is informed whenever they best their previous high score.
- **Replay Game** – the user is prompted at the end of a game if they wish to replay. This enhances user experience as they do not need to reload the game.
- **Options Menu** – by typing 'c' a list of additional commands are displayed which allows the user to check move number, reveal the target total for each row, game play instructions or open a help menu explaining how to enter a move.
- **Quit Option** – the user can quit the game at any time by entering q.
- **Clear Screen** – prints 100 empty lines to give the illusion there is only one grid being updated

Solution Quality

The final solution utilized abstraction, encapsulation, polymorphism, composition and aggregation to deliver a modular, highly abstract, extendible solution. A single instance of GameManager can start new games with a single method call and each game will give the user the option to replay the game at the end of each play through.

The solution is appropriately commented and each method has a javadoc explaining its use. Field variables are private with appropriate access provided through accessors. Function overloading is used to allow assignment of magic square values by grid position (`int[]`) or coordinates (`int x, int y`).

The solution also pays attention to time complexity with the exception of reading/ writing the 2D array, no nested for loop are used meaning the methods run at linear time or faster.

Error handling is performed through try-catch blocks with appropriate user feedback. By using composition, the code is easy to maintain as methods are stored with their corresponding class e.g. all UI logic will be found in the UIManager class.

Interesting Results

One important aspect of playing a game is user experience. To enhance delivery I chose a fantastical approach wrapped in a colourful ASCII wrapper. The user is whisked away to a magic castle where a wizard stores his treasure with magic square locks. By solving the grids the user is rewarded with ASCII goodness.

The actual ASCII art was simple as there are many programs which will convert images to ASCII, my wizard avatar utilized one such program (Fig 3.1).

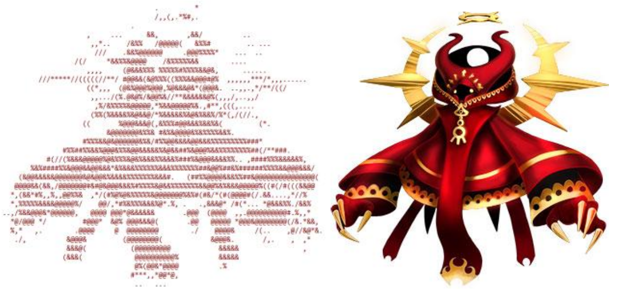


Fig 3.1 – ASCII art conversion of wizard avatar

The ASCII for the text was a little more difficult as it is programitcally generated. I utilized some existing code[4] but had to make some adjustments to allow it to be used for different word sizes, utilize ANSI colours and accept an offset.

The art, colours and story elements all offer a level of polish that improves the user experience.

Difficulties

The initial construction of the grid, specifically wrapping, took longer than I would have liked. There were a few solutions that came to mind to handle wrapping, including using modulus as well as not creating separate variables but both were dismissed as the resultant code was not logical to follow. The implemented version - using wrappedX and wrappedY to act as pointers to see if the grid slot is available – offered clear logic on what was going on and its superior readability outweighed minimal efficiency gains.

Input Validation

User input was validated through try-catch blocks and for move inputs, `trim()` and regex expressions were used. By using different exception blocks, the user received tailored feedback e.g. for wrong moves they would get: "Not a valid direction" or "Move must have 3 parts" rather than a generic "wrong move".

Future Additions

- Implement an improved shuffling algorithm that mixes the grid better
- Add the ability to reset the grid
- Undo move button

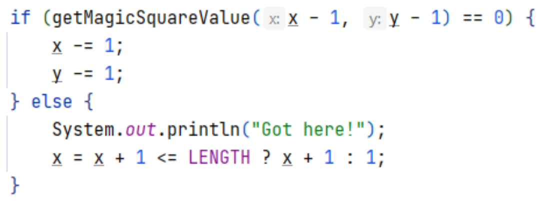
Software Testing Methodologies

Software errors, broadly speaking, fall into three categories: syntax errors, exception errors and logic errors. The javac compiler will handle syntax errors so code validation for this project addressed the other two points using a four pronged approach:

- Debugging code using the print statements
- Debugging code using a debugger
- Unit testing public methods
- User testing by playing the game

System.out.Println

One of the fastest solutions to solve exception and logic errors is adding `println()` statement before and after where the error has occurred. When first constructing the magic square I had a logic error where the grid was not being constructed correctly. Adding a print statement to the else logic (Fig 4.1) quickly revealed there was an error in the if else logic as the “Got here” statement never printed.



```
if (getMagicSquareValue( x: x - 1,  y: y - 1) == 0) {
    x -= 1;
    y -= 1;
} else {
    System.out.println("Got here!");
    x = x + 1 <= LENGTH ? x + 1 : 1;
}
```

Fig 4.1 – Println statement to debug logic error

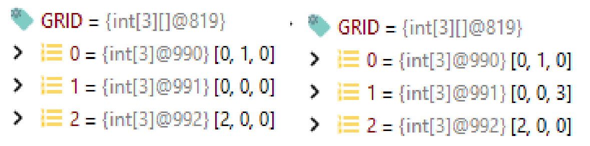
The solution was to use wrappedX and wrappedY as pointers to check grid position before inserting.

Built-in Debugger

When logic errors are more complicated, built-in debugging tools can be extremely useful.

A different grid error, one where the constructed grid did not match the question sheet example grid, was not solvable by print statements. Instead, using the debugger, I placed a breakpoint just before the for loop which inserts numbers into the grid and then started a debug session.

As I stepped through the program I was monitoring the GRID and you can see how before (Fig 4.2 left) and after (Fig 4.2 right) a 3 is inserted at (2, 3). This allowed me to follow the grid construction and helped pinpoint exactly which i-value was causing the problem and then allowed me to make the necessary corrections.



```
GRID = {int[3][]@819}
>  0 = {int[3]@990} [0, 1, 0]
>  1 = {int[3]@991} [0, 0, 0]
>  2 = {int[3]@992} [2, 0, 0]
```

```
GRID = {int[3][]@819}
>  0 = {int[3]@990} [0, 1, 0]
>  1 = {int[3]@991} [0, 0, 3]
>  2 = {int[3]@992} [2, 0, 0]
```

Fig 4.2 – IntelliJ IDEA debug output pre and post insertion

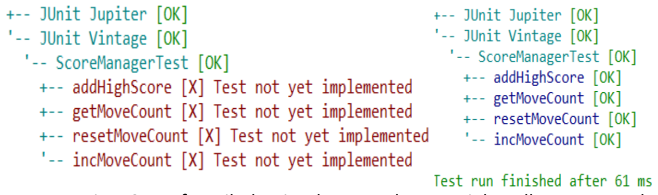
Unit Testing with Junit

Testing of class methods was done using JUnit. JUnit allows one to write code with an expected outcome and then test it against a method whose output is compared with it using assertions.

Following Microsoft’s best practices for writing unit tests[5], tests were made using a AAA approach:

Arrange – set up the test conditions
Act – produce the expected and actual values
Assert – check if the values produced in Act match

Only public methods (excluding accessors and mutators) were tested to reflect actual usage. Empty tests methods were written resulting in failures (Fig 4.3 left) before solutions were implemented and the tests rerun (Fig 4.3 right).



```
+-- JUnit Jupiter [OK]
'-- JUnit Vintage [OK]
  '-- ScoreManagerTest [OK]
    +-- addHighScore [X] Test not yet implemented
    +-- getMoveCount [X] Test not yet implemented
    +-- resetMoveCount [X] Test not yet implemented
    '-- incMoveCount [X] Test not yet implemented
```

```
+-- JUnit Jupiter [OK]
'-- JUnit Vintage [OK]
  '-- ScoreManagerTest [OK]
    +-- addHighScore [OK]
    +-- getMoveCount [OK]
    +-- resetMoveCount [OK]
    '-- incMoveCount [OK]

Test run finished after 61 ms
```

Fig 4.3 – Left: Failed unimplemented tests. Right: All tests passed

A few problems were encountered during the testing phase. The first was the GameManager class was so well encapsulated that it was impossible to test its logic directly. This was solved by creating a DebugGameManager class which exposed its methods as public but only change the constructor so not to start the game. The main disadvantage of this solution is the fact that two copies of the same class need to be maintained, the advantage is the original GameManager class exhibits textbook encapsulation and abstraction.

The next issue was having to repeat code to test different sized Magic Squares. This was addressed by decomposing the test from MagicSquareTest into a parameterized test class ensuring efficient DRY code.

Another issue was handling print statements. Reading the JUnit documentation revealed a method[6] of using PrintStream. The tests had to be tweaked to remove the ANSI colour encodings but otherwise worked as expected.

The final issue was testing the shuffle method – randomness is difficult to test. A seed was considered but it still wouldn’t solve if the grid was shuffled only up/ down and left/right. So a hybrid approach was used where a shuffled grid was tested against a non shuffled grid and then using the debug method in fig 4.2, I manually checked which numbers were being swapped.

User Testing

The unit tests provided a thorough foundation for testing code logic but one area it failed to address was testing user input. This is where user testing came in where I tried to break the game by entering invalid inputs at every opportunity.

One unresolved issue related to grid sizes larger than the terminal width causing overlap. I found no good way to resolve this because even if I used terminal width, I found no solution which updates terminal width in real time should the user resize their terminal.

Overall, the combination of testing approaches used, hopefully, resulted in a bug free user experience.

References

[1] Yoke, Ho Peng. 2008. *Magic Squares in China*. Encyclopedia of the History of Science, Technology. Springer. pp. 1252–1259.

[2]Lin J. 2020. *Magic Square 3x3*. Available at: vvlnote.github.io/magic_square_3x3 [Accessed March 2020]

[3] Higgins, P. 2008. Number Story: From Counting to Cryptography. New York: Copernicus. p. 5

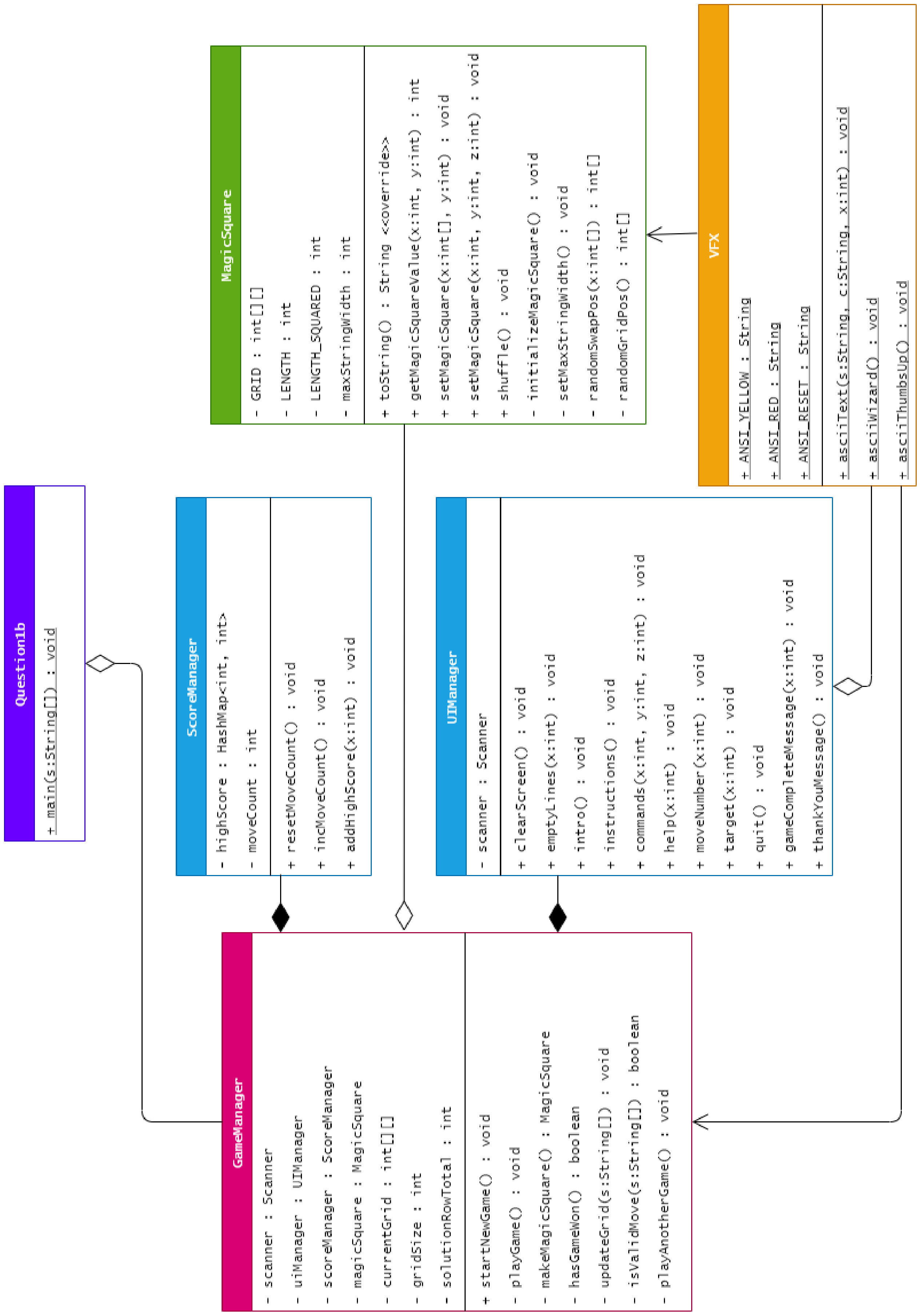
[4]Mykong. 2017. *ASCII Art Java example*. Available at: mkyong.com/java/ascii-art-java-example [Accessed March 2020]

[5]Reese J.2018. *Unit testing best practices with .NET Core*. Available at: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices [Accessed March 2020]

[6]*JUnit 5 user guide*. Available at: junit.org/junit5/docs/current/user-guide [Accessed March 2020]

Appendix

Appendix 1.1 – Magic Square game solution full UML (Unified Modelling Language) class diagram


MagicSquare
- GRID : int[][]
- LENGTH : int
- LENGTH_SQUARED : int
- maxStringWidth : int
+ toString() : String <<override>>
+ getMagicSquareValue(x:int, y:int) : int
+ setMagicSquare(x:int[], y:int) : void
+ setMagicSquare(x:int, y:int, z:int) : void
+ shuffle() : void
- initializeMagicSquare() : void
- setMaxStringWidth() : void
- randomSwapPos(x:int[]) : int[]
- randomGridPos() : int[]
VFX
+ ANSI_YELLOW : String
+ ANSI_RED : String
+ ANSI_RESET : String
+ asciiText(s:String, c:String, x:int) : void
+ asciiWizard() : void
+ asciiThumbsUp() : void
Question1b
+ main(s:String[]) : void
ScoreManager
- highScore : HashMap<int, int>
- moveCount : int
+ resetMoveCount() : void
+ incMoveCount() : void
+ addHighScore(x:int) : void
UIManager
- scanner : Scanner
+ clearScreen() : void
+ emptyLines(x:int) : void
+ intro() : void
+ instructions() : void
+ commands(x:int, y:int, z:int) : void
+ help(x:int) : void
+ moveNumber(x:int) : void
+ target(x:int) : void
+ quit() : void
+ gameCompleteMessage(x:int) : void
+ thankYouMessage() : void
GameManager
- scanner : Scanner
- uiManager : UIManager
- scoreManager : ScoreManager
- magicSquare : MagicSquare
- currentGrid : int[][]
- gridSize : int
- solutionRowTotal : int
+ startNewGame() : void
- playGame() : void
- makeMagicSquare() : MagicSquare
- hasGameWon() : boolean
- updateGrid(s:String[]) : void
- isValidMove(s:String[]) : boolean
- playAnotherGame() : void