SWEN30006 Project 1 - Tetris

Part 1: Constructing a domain model.
To develop a visual understanding of the problem domain, we read the full specification of the game and how it needed to be extended. From there we constructed our domain model making various decisions on new classes and where to assign responsibilities.

Part 2: Design Model and Implementation
The original code had a lot of problems that capped its extensibility as it didn't follow a lot of OO principles. Below is a detailed outline of what changes were made and why they were to have medium and madness levels added, and also the stats manager.

Making a letter superclass
There was a class for every single type of block but there was no parent class. This means that every class had to rewrite the same code. This duplicates the code in an unnecessary way. For all of the individual TetroLetter classes, they share the same functions, namely rotate(), left(), drop(),etc. From a design standpoint, this violates the inheritance concept of Object Oriented Programming.

We created a superclass called TetroLetter that has all the methods, the subclasses inherit all the movement methods but set their own rotation coordinates, blockId and blockName as each block is unique. This allows us to reuse code and lets us extend our pieces without having to change the Tetris class (as it relies on the letter superclass). The TetroLetter class also allowed us to significantly shorten our Tetris.moveBlock(), blockPreview(), autoBlockMove() functions as when the currentBlock is called, it doesn't have to be downcasted.

Additionally, we want to be able to extend our program to be able to use pieces with 5 blocks. Although the new "P, Q, Plus" shaped pieces have 5 blocks, they still behave the same way with user input and have 4 orientations and therefore are able to use our letter superclass. The Letter superclass is the stable class and therefore the other variations (may have 4, 5 or even 6 blocks) do not have an impact on the overall superclass and therefore won't affect the behaviour of the overall game. This is why when there are 5 blocks the rotation array is simply overwritten to be an array of 5 TetroBlocks instead.

Reading Difficulty
We decided to make the Driver in charge of reading what difficulty the game had to be instead of the Tetris class. If the Tetris class had to set its own difficulty there would be a lot of "if difficulty == xxx" statements that cause lower cohesion and high coupling. The Driver class takes the difficulty as a String and depending on the string instantiates the appropriate Tetris subclass.

Abstracting Tetris
We decided to make the Tetris class abstract and have various variables and functions protected therefore subclasses are allowed to override them yet can not affect other subclasses. This allows the respondent to be spread among the subclasses and makes sure that new modes can not affect each other. The Tetris class is abstract with 3 subclasses called Easy, Medium and Madness. The protected variables included the numBlocks, getSlowDown(), moveBlock(). The int numBlocks was replaced as the max for "rnd" saying how many rnd = random.nextInt(numBlocks). Although the different classes had related variables, if they inherited each other there would be an over-dependance. We also added a function called previewBlock(TetroLetter p) to reduce code repetition.

Difficulty Easy:

We still had to define the specifications for Easy that separated it from the other levels. The easy subclass specified the number of numBlocks to be 7 as it only has the 7, 4 blocked TetroLetters.

Difficulty Medium:
We set the numBlocks to be 10 and the medium difficulty includes the 7, 4 blocked TetroLetters and the 3, 5 blocked TetroLetters. In the original Tetris code, every random tetroLetter setSlowdown() was controlled when the random block spawned. In order to be able to manipulate the slowdown speeds we created a getSlowDown() method that simply returned the slowDown value for Easy, since this was a protected it function we then overwritten it to return the slowDown/1.2 to make it 20% faster (because slowDown is inversely proportional to Speed). This means that every time a new TetroLetter is instantiated, it calls a function to return the slowDown appropriate to the difficulty.

Difficulty Madness:
We set the numBlocks to be 10 and the medium difficulty includes the 7, 4 blocked TetroLetters and the 3, 5 blocked TetroLetters. We overwrote the getSlowDown() to return a random integer between slowdown/2 and slowdown. The moveBlock() function was also overwritten to only take input from the left, right and down keys.

Although three difficulty levels can vastly be different, they still share the same basis. From the polymorphism concept of OOP, we will be implementing the Protected Variation pattern by creating a stable abstract Tetris class and different levels inherit the original code. Each class sets its own set of valid blocks, the speed (if the speed is random the speed is a function that generates a random number) and whether or not rotate is on.

Printing Stats
Every time the player presses the “start” button, the game will record the previous round’s stats and update the Statistics files. We created a StatManager class that is instantiated by the Tetris class. The Tetris class creates 1 StatManager and sends info to it to handle the stats of the game. First stat.setDifficulty() is set by the subclasses, everytime a new t block is spawned it is sent via stats.updatePiece(t.blockName) (function that adds tallys the piece on a hashmap), and when a new rounds starts it calls stats.update(). When stats.update() is called the currentRound, totalScore, and the formatting for allRounds is updated and then the hashmap is reset. The old Statistics.txt file is deleted and then rewritten with the new information.

Although three difficulty levels can vastly be different, they still share the same basis. From the polymorphism concept of OOP, we will be implementing the Protected Variation pattern by creating a stable abstract Tetris superclass and different levels inherit the original code. Each class sets its own set of valid blocks, the speed (if the speed is random the speed is a function that generates a random number) and whether or not rotate is on.

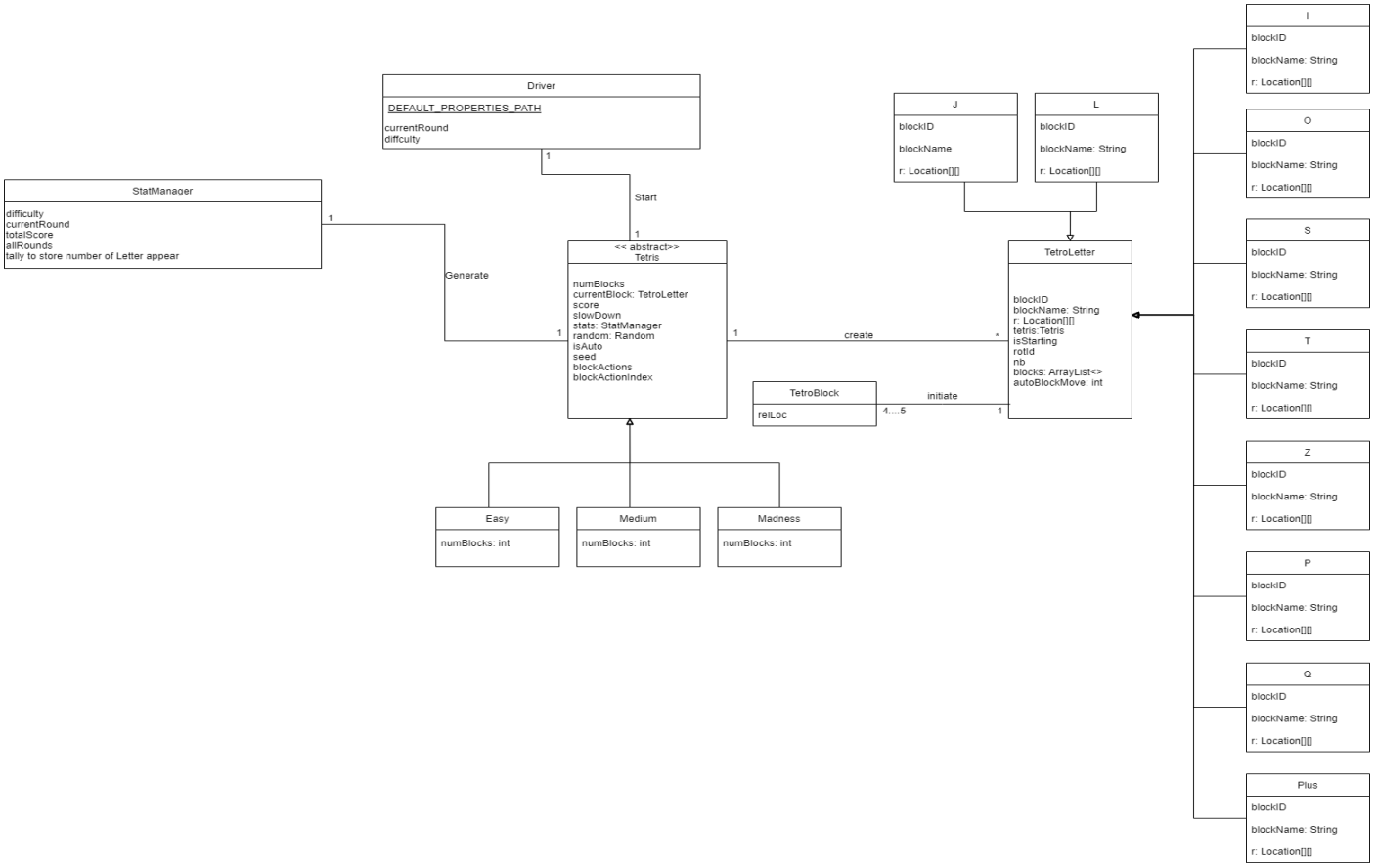

Figure 1: Partial domain diagram describing the relation of each class of Tetris Game

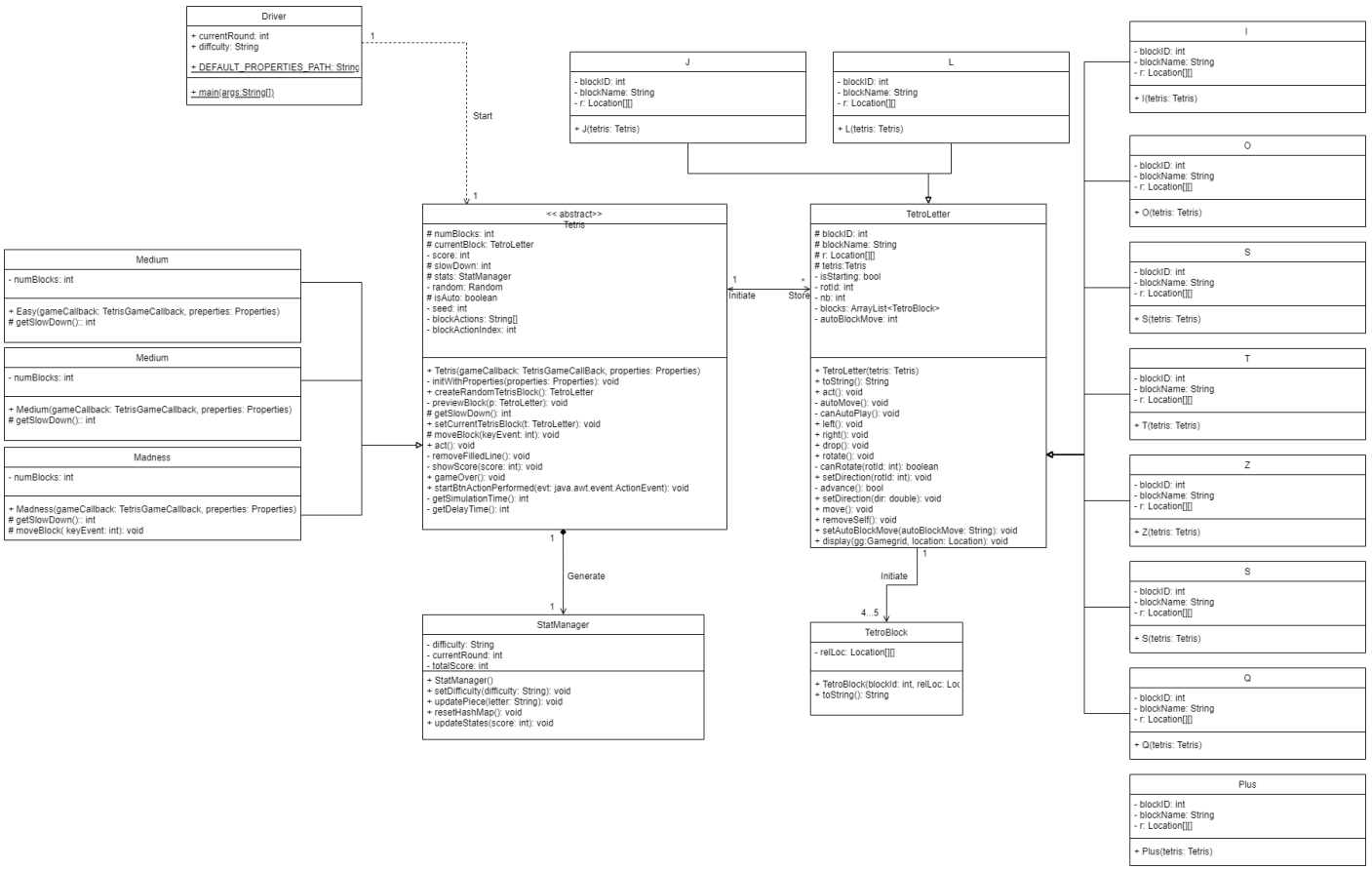

Figure 2: a Design Class Diagram of the Tetris classes and elements. Note that it is a partial diagram that focuses on the implementation of the game and the changes made

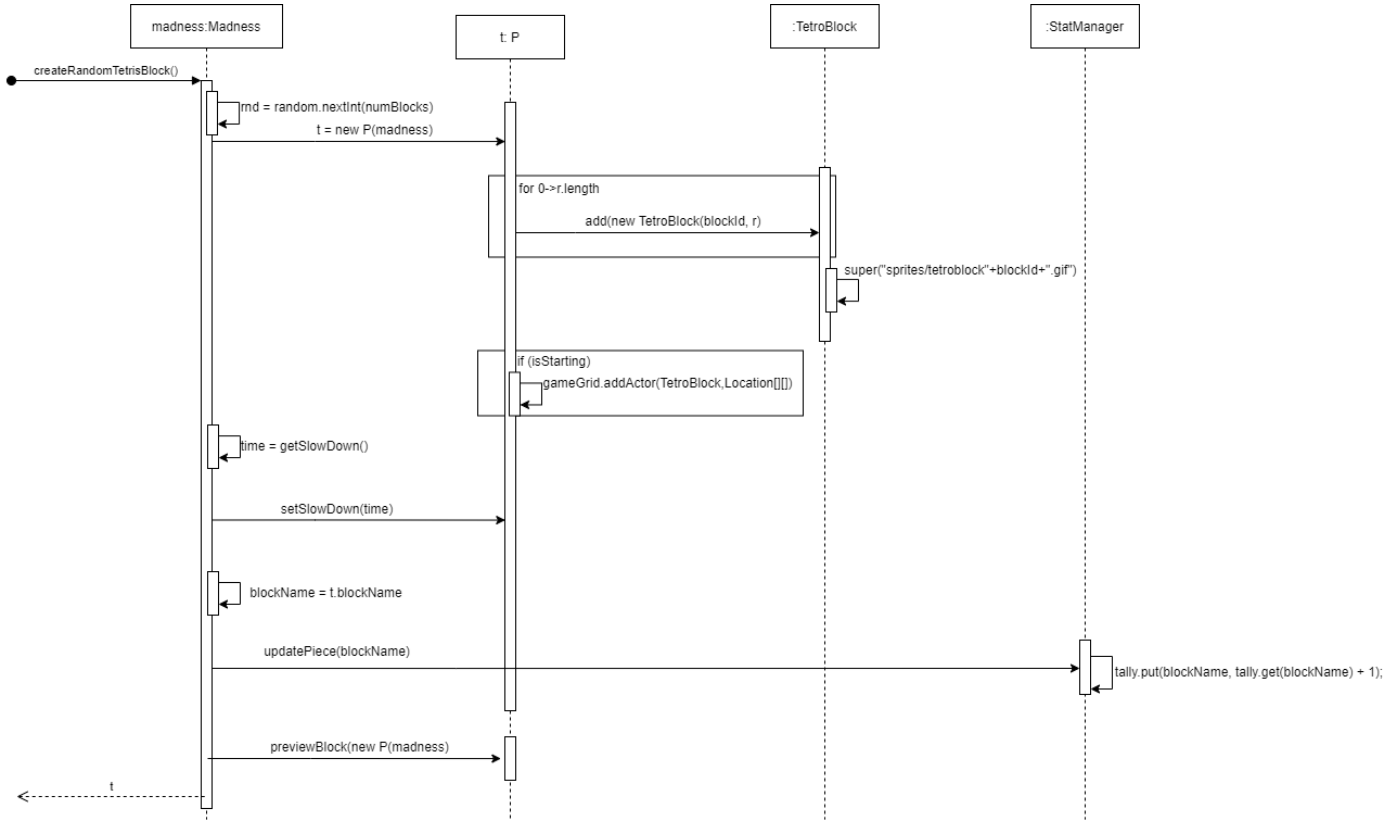

Figure 3: A Sequence diagram describing the system's behaviour for one falling shape with difficulty level as "madness", no keyboard actions from the user and statistic recording is enabled.