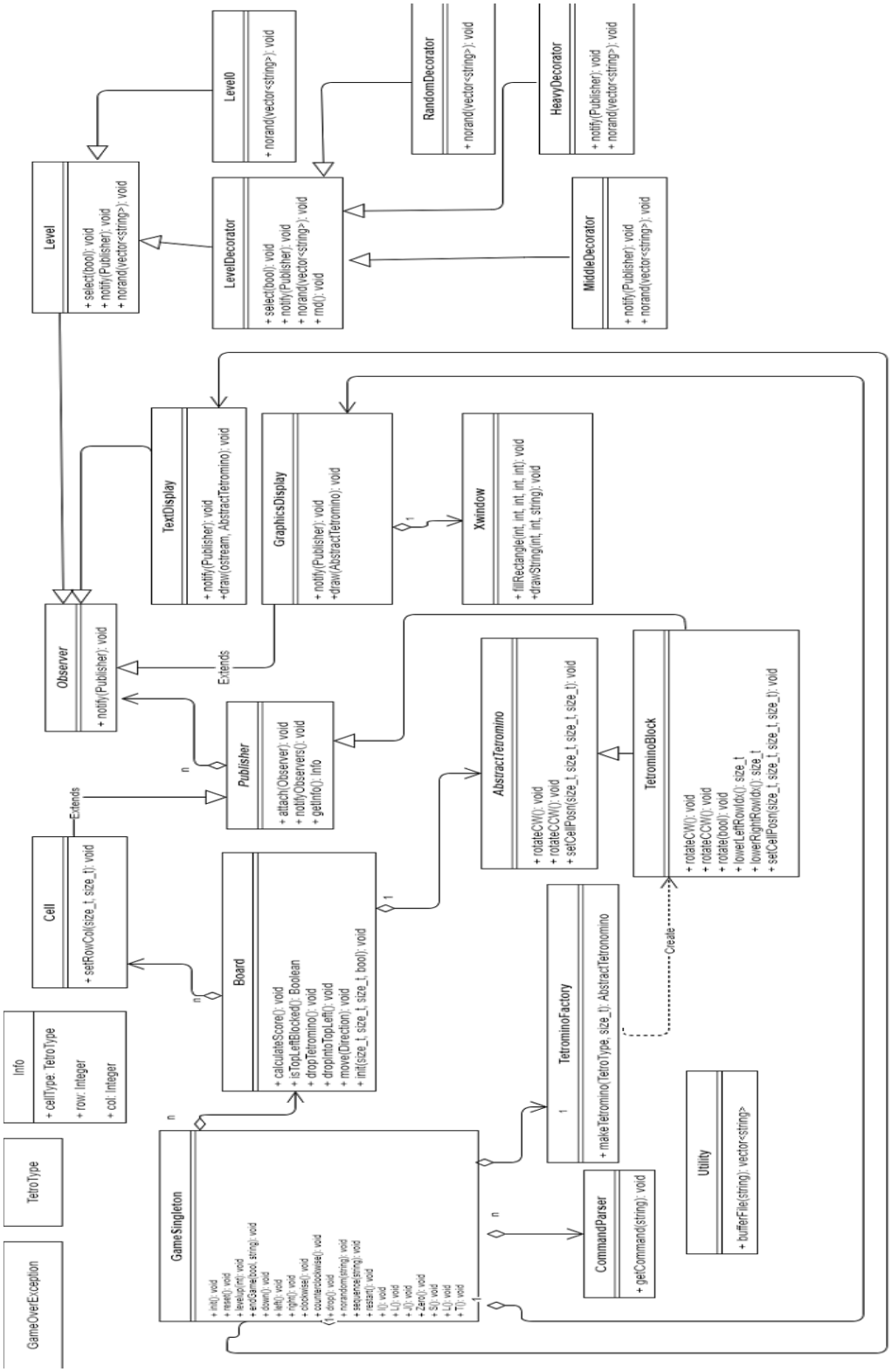
Overview

This report is on the project Quadris. Our program directory structure is:

```
/tests
    ….
/src
    /h
        ...
    /cc
        ...
```

/src contains the subdirectories /h and /cc, which contan the .h fils and the .cc files respectively. The program can be divided into the following general modules: Game, Board, Command Parser, Tetromino generator, Level decorator, and UI displays. The relationships between these modules is shown in the UML section.

Updated UML


GameOverException
TetroType
Info
+ cellType: TetroType
+ row: Integer
+ col: Integer
Level
+ select(bool): void
+ notify(Publisher): void
+ norand(vector<string>): void
Level0
+ norand(vector<string>): void
LevelDecorator
+ select(bool): void
+ notify(Publisher): void
+ norand(vector<string>): void
+ rnd(): void
RandomDecorator
+ norand(vector<string>): void
HeavyDecorator
+ notify(Publisher): void
+ norand(vector<string>): void
MiddleDecorator
+ notify(Publisher): void
+ norand(vector<string>): void
TextDisplay
+ notify(Publisher): void
+draw(ostream, AbstractTetromino): void
GraphicsDisplay
+ notify(Publisher): void
+draw(AbstractTetromino): void
Xwindow
+ fillRectangle(int, int, int, int): void
+drawString(int, int, string): void
Observer
+ notify(Publisher): void
Extends
Publisher
+ attach(Observer): void
+ notify(Observers(): void
+ getInfo(): Info
Cell
+ setRowCol(size_t, size_t): void
Board
+ calculateScore(): void
+ isTopLeftBlocked(): Boolean
+ dropTetromino(): void
+ dropIntoTopLeft(): void
+ move(Direction): void
+ init(size_t, size_t, bool): void
AbstractTetromino
+ rotateCW(): void
+ rotateCCW(): void
+ setCellPosn(size_t, size_t, size_t, size_t): void
TetrominoBlock
+ rotateCW(): void
+ rotateCCW(): void
+ rotate(bool): void
+ lowerLeftRowIdx(): size_t
+ lowerRightRowIdx(): size_t
+ setCellPosn(size_t, size_t, size_t, size_t): void
Create
TetrominoFactory
+ makeTetromino(TetroType, size_t): AbstractTetronomino
GameSingleton
+ init(): void
+ reset(): void
+ levelup(int): void
+ endGame(bool, string): void
+ down(): void
+ left(): void
+ right(): void
+ clockwise(): void
+ counterclockwise(): void
+ drop(): void
+ norandom(string): void
+ sequence(string): void
+ restart(): void
+ I(): void
+ J(): void
+ L(): void
+ O(): void
+ Z(): void
+ S(): void
+ J(): void
+ T(): void
CommandParser
+ getCommand(string): void
Utility
+ bufferFile(string): vector<string>
Extends

Design

The Board object contains a vector of vectors of Cell objects, which each have information on the type and ID of block that Cell came from. Whenever a block is dropped, it checks each row to see if that row has no None type cells in it, which means that it is full. In this case, the row is cleared, and all rows above it move down by one, which is done by setting the below row to the row above, and setting its cell location row down one. Since we have the block ID stored in the cells, then we can check for each ID that has been dropped, whether it still exists on the board, and if not, give points for clearing a full block.

We used the Observer design pattern. We have both the TextDisplay and GraphicsDisplay classes observe the GameSingleton object and all Cell objects. The TextDisplay and GraphicsDisplay publish to GameSingleton when the score, hiscore, and level change. TextDisplay and GraphicsDisplay observe the Cells which tell the displays what colour and character to put at the Cells' positions in the display.

We used the Factory design pattern. We have a TetrominoFactory that makes TetrominoBlock objects, and returns shared pointers to them. This is used to generate the next tetromino, and then it is stored until the previous one is dropped and it becomes the new current tetromino.

For the levels, we wanted to be able to construct them from text files or command input so we used a decorator design pattern. The base level is the one that parses from file and you can add decorators on top such as the heavy decorator, middle block decorator and the random number decorator. This allows flexibility in construction of levels and allows for text file definitions of levels such as double heavy levels. For the random decorator we construct the probabilities of each piece from the level text file.

For the Command parser we used a vector of lambda functions which affords us the flexibility of easily adding new commands and macro functionalities as we can programmatically expand or reduce the amount of commands and construct new commands and macros at runtime assuming it does not use unknown functionalities.

For the central game we used a singleton pattern primarily for ease of programming as static access to the instances is very convenient and allows for stateless calls to it used by the lambda functions in the command parser. There are also a fair share of passthrough functions that access private members of the gamesingleton. In a way its a model view controller system where the the gamesingleton acts as a software interface between the model represented by the board, the commandparser is the controller used by the user and the graphics and text displays are the views that are updated by the model and displayed.

For restarting the game after a block cannot be spawned or the restart command is called, we had it so the game saves the hiscore in a file, and then reloads the program. This has the added bonus effect of letting us carry over the hiscore from previous game runs.

Resilience to Change

Low coupling is achieved and is discussed below. In general, within reason, big changes to the specification can be handled by the program. Changes like removing the file reader without providing another way of reading from files or another storage method will break the program. However, swapping out file reading for a database query is feasible as long as the new utility returns a vector of strings of the input. This is true for all inputs and outputs - modules that read or return something do not care about the implementation details.

Suppose a level needs to be made or deleted. Since we are reading in from a file the parameters of all the levels, it is possible to add and remove from that file without recompiling. Moreover if all levels defined in the text file are removed, then the default level 0 is still functional.

Suppose a new type of block is created, then you need to define it in the enum class in h/info.h, and write the necessary setCell(row, col) lines in the cc/tetrominofactory.cc. We acknowledge it would be more general to read these shapes from a file. If we did that then it would be possible to create new blocks by editing a file and adding to the enum class rather than knowing what setCell(row, col) does.

Suppose the formula for calculating score changes. Then, within reason, we could just change the formula in cc/board.cc. If the axis of rotation of blocks changes, then change the formula that calculates rotationRow and rotationCol in TetrominoBlock::rotate() if the new formula uses information we already store.

Suppose instead of rows being cleared, columns needed to be cleared. Our board and tetromino blocks are represented with vectors of vectors of Cells. The first dimension of the vector is currently representing the rows and the second dimension are the columns since this layout makes it very simple to clear rows by clearing and reassigning vectors at indices. To clear columns, flip the dimensions so that the first is the columns and the second the rows. Then change the necessary variables names in board and other classes such as in TetrominoBlock::rotate() which iterates through the first dimension and names each vector at the indices as row and rowTemp. This indicates that low cohesion is still achieved in this respect - a major system revamp is not needed. Only a change in variable names is needed.

Furthermore, and in detail, to achieve high cohesion, each class also only executes its own purpose:

- Class AbstractTetromino: represents everything there is to know about a tetromino relating to its identity, size, what its Cells are, and where it is on the board. Where it is on the board is determined through its component cells.
- Class TetrominoBlock is the concrete implementation of the AbstractTetromino class. It also rotates its block representation.
- Class Board knows everything there is to know about its grid. It can clear rows, move the current tetromino regardless of the tetromino type, it can check for collisions upon

rotations and movements. The board needs to know about the current tetromino because it needs to move it. To check for rotation collisions, it first rotations a copy of its current tetromino. Then if it's possible, then it actually rotates its tetromino. The vector of the board contains Cells, not tetrominos so the only tetromino Board knows about is its current tetromino. It also does not know the next tetromino.

- Class Cell: knows its type and what its row and column are.
- Class GameOverException: only deals with defining an exception along with a message.
- Class GameSingleton: is a fairly broad class but it is required to be broad since it is the central game object responsible for responding to input commands.
- Class GraphicsDisplay and TextDisplay both only receive input as their are observers and display output.
- Class CommandParser only reads input and calls the corresponding function in GameSingleton or creates a macro command.
- Observer and Publisher are each for the Observer pattern
- Class TetrominoFactory only makes a block when given a block type to make. It knows nothing about the current level.

Answers to Questions

Question 1

1. How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen?

   Each cell block has an ID corresponding to the tetromino it came from. Every time a new tetromino is generated its ID is set to previousID + 1. Once the ID of the new tetromino is a 10 above the ID of a cell already in the board, it would destroy that block.

2. Could the generation of such blocks be easily confined to more advanced levels?

   A level with this rule can be created and code that removes tetromino blocks based on this can be written in the Game singleton. The gamesingleon.cc would then need to call a command in Board to delete the blocks.

Question 2

1. How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

   You would create a Level class, with each of the different levels as objects of the class. The class has decorator parameters: the tetromino block probabilities, whether or not it is random, whether or not blocks are heavy, and so on. In the gamesingleton object

there would be a vector of the different level objects. To add a new level, make a new Level object, apply the decorators, and add it to the vector.

Question 3

1. How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename counterclockwise cc)?

   To make a new command, you can add a new line to the map of commands in cc/cmdparser.cc and a corresponding function in gamesingleton.cc and gamesingleton.h. The corresponding function would execute the necessary block of code, either directly in gamesingleton.cc or it would call another function in another part of the program).

   To change a command name, simply type "`macro [NAME] [EXISTING COMMAND]`" at runtime.

   Changes to existing command names are handled by the command parser which has a function that makes the necessary changes to the map of commands from strings to function pointers. That is, if you make a macro for counterclockwise named cc, the lookup map finds the function for counterclockwise and also maps cc to that function.

2. How might you support a "macro" language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

   To name a sequence of commands, type "`macro [NAME] [EXISTING COMMANDS]`" at runtime.

   To make shortcuts to chains of commands, we use a map. It maps from a shortcut string, to a string of commands that the shortcut corresponds to. When the user enters a new shortcut and the commands it represents, it checks to see if the desired shortcut does not already exist as a command, and then if it doesn't, adds it to the map. If the command parser detects that you've entered one of the shortcut strings, it executes the commands mapped to it by iterating through the string of commands and calling each corresponding function in gamesingleton.cc.

Extra Credit Features

Creating different names for commands and creating a shortcut to a sequence of commands are explained above in the answers to questions 3.1 and 3.2. No recompilation is necessary for these features.

Final Questions

1) What lessons did this project teach you about developing software in teams?

   We learned that often each person only knows how their part functions so everyone has to somehow make each person's part communicate correctly with other people's parts. Using design patterns helps with this challenge. For example, since we used the Observer pattern to communicate between the board and the text display the people implementing those components just had to know how our observers and publishers work. So, no one had to know the details of everyone's code. They just had to know what it can accept and return.

   Furthermore, it is important to make sure everyone's parts put together compile. Only pushing code to the repository that compiles is usually a good idea. Testing often is also important so that we know the parts function as expected.

2) What would you have done differently if you had the chance to start over?

   We would create tests from day 1 before DD1 and run them as we wrote the program. It would be nice to use continuous integration to make sure everyone's working parts integrate with each other and pass the tests. We would also write unit tests for individual functions like rotate clockwise without printing the entire board. This would mean that we would know specifically if rotations work without rotations depending on a correct board implementation. Specifically for rotations, an .in and .out file could respectively look like:

   ```
   clockwise
   J
   JJJ
   ```

   ```
   JJ
   J
   J
   ```