Design document: Quadris

Our game, Quadris, is run by executing the file `quadris`, and can take commands before it is run on the command-line. As specified in the project guideline, our program can take the following commands: `-text`, `-seed xxx`, `-scriptfile xxx`, `-startlevel n`. Our program starts by calling the main method in main.cc. This method is what handles the command-line options as stated previously, setting variables accordingly, such as `bool GUI = false;` for `-text`.

After checking for these command-line options, the main method makes an instance of `Game`, providing it with the level, seed value, whether GUI is on, and optionally, the file name of the level with predetermined blocks. In `Game`'s constructor, an instance of the `Board` class is created, an instance of the `RandomBlock` class is created, and a vector of `Block`s is created. We will discuss these components in the following paragraphs.

The `Board` class accurately describes the main playing field of Quadris, which is a board. The board contains a 15×10 2D-array of `Cell`s, as specified in the project guideline. The Board is important for two main purposes. Firstly, it handles the relationships between the `Cell`s, allowing easy access to a `Cell`'s neighbours as long as you know its X and Y values (in regards to the 2D-array). Secondly, it handles displaying the contents of the Quadris board in both text and GUI form by `operator<<` and `XwindowUpdate` methods respectively.

The `Block` class contains information of each type of Quadris block (e.g. Z block, L block). The information that it includes is the character used to represent it on the text display, the `Cell`s that are components of the `Block`, the colour of the `Block` for the GUI, and what the `Block` looks like in pure string form (in contrast to cell form). The methods `shift` and `rotate` are in the Block class, which are responsible for handling the movement of blocks around the board. This essentially consists of creating a new array of `Cell`s consisting of the block's new location, checking if these Cells are free, and then reassigning values to the involved `Cell`s and updating the components of the `Block`. As well, the movement methods also call the `Board`'s `XwindowUpdate` function to update the GUI of the `Cell`s' new states. The class also contains a function `notify` which is called by component Cells when they are cleared. This method tells the `Block` to decrement the number of living `Cell`s by one. Finally, the class has an `operator<<` method which provides the output stream with the shape of the `Block` without being on the `Cell`. This is useful for showing the next block.

The `RandomBlock` class is the class used to handle the creation of Blocks based on a sequence or probability provided. This is based on whether a file name was provided for a sequence and the level number. On construction, the RandomBlock class reads the file `block.txt`. This is the file that contains the information about the number of blocks, the block colours, and the block shapes. The following is an excerpt from the default `blocks.txt` we are using.

```
7
5
IIII
4
J
JJJ
```

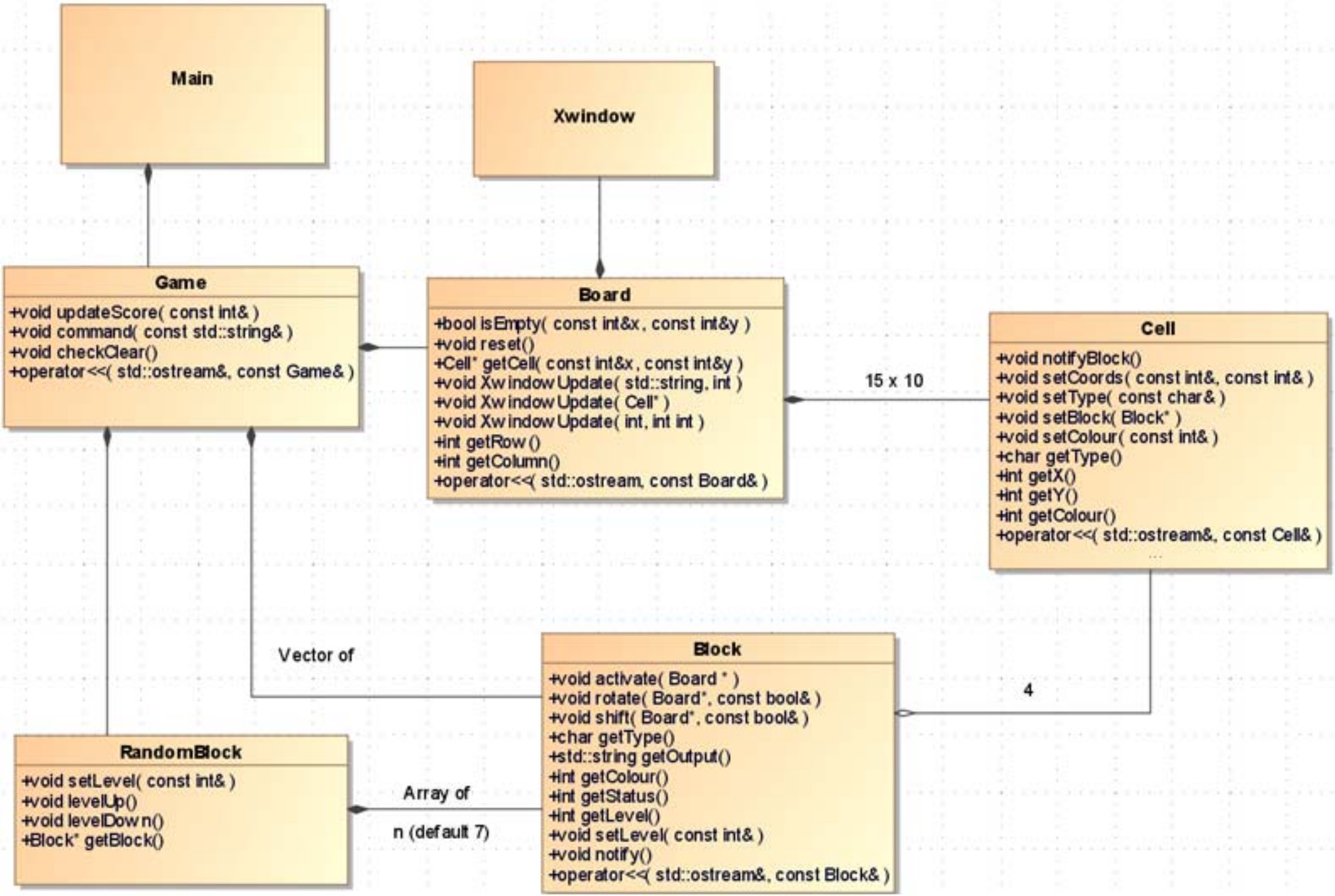
The first line consists of a single integer, *n*, which is 7 in the above example. This tells the number of Blocks. This is followed by *n* of following pattern: *c*, the colour integer, followed by the block in text format, made up entirely of the same character.

`RandomBlock` reads this file and creates an array of `Block*` called `origBlocks`. This array has a length of *n*. The *n* `Block`s are the blocks described in `blocks.txt`. The four `Cell`s provided to each `Block` are all based such that the block will be leftmost and leave three rows above it empty in the Quadris board.

`RandomBlock` provides a method called `getBlock`. This method randomly generates an integer, and chooses a `Block` type to return based on the number. The probability of the type of block is based on the level. If the level is 0 and a file is provided, the method will instead read from the file. When the choice is made, `Block`'s copy constructor is called, and the corresponding `Block` in `origBlocks` is copied and returned. It is also in the copy constructor where the Block called `activate()`, which notifies the `Block`'s component `Cell`s of its existence and to set themselves and change colour accordingly. This will also call `XwindowUpdate` in Board to update the GUI.

Mentioned earlier was a `Block` vector in the `Game` class. This vector keeps track of all of the `Block`s that currently exist on the Quadris board. If all of the component `Cell`s of a `Block` are no longer alive, it will be removed from the vector and the score will be updated. Furthermore, the very last `Block` on the vector will always be the "next block".

When a Quadris block is dropped or is commanded to move down and cannot move any further, the block is set. Every time a block is set, there is potential for a line to be complete. So, each row of `Cell`s in `Board` is checked. If a row contains no cells with type ' ' (space), then a row is completely filled and must be cleared. The `Cell`s are then shifted downward by swapping the addresses in the 2D-array in `Board`. The score is then updated to reflect the line clear. As well, all of the `Cell`s that were cleared notify their respective `Block`s, and the integer `alive` is updated accordingly. Then, traversing the vector of `Block`s in `Game`, any `Block` that has `alive==0` is removed from the vector. The score is updated accordingly.


Main
Xwindow
Game
+void updateScore( const int& )
+void command( const std::string& )
+void checkClear()
+operator<<( std::ostream&, const Game& )
Board
+bool isEmpty( const int&x, const int&y )
+void reset()
+Cell* getCell( const int&x, const int&y )
+void XwindowUpdate( std::string, int )
+void XwindowUpdate( Cell* )
+void XwindowUpdate( int, int int )
+int getRow()
+int getColumn()
+operator<<( std::ostream, const Board& )
15 x 10
Cell
+void notifyBlock()
+void setCoords( const int&, const int& )
+void setType( const char& )
+void setBlock( Block* )
+void setColour( const int& )
+char getType()
+int getX()
+int getY()
+int getColour()
+operator<<( std::ostream&, const Cell& )
...
Vector of
Block
+void activate( Board * )
+void rotate( Board*, const bool& )
+void shift( Board*, const bool& )
+char getType()
+std::string getOutput()
+int getColour()
+int getStatus()
+int getLevel()
+void setLevel( const int& )
+void notify()
+operator<<( std::ostream&, const Block& )
4
RandomBlock
+void setLevel( const int& )
+void levelUp()
+void levelDown()
+Block* getBlock()
Array of
n (default 7)

Differences from Due Date 1

The biggest difference in our design for the program compared to our plan on due date 1 is the revamp of the way we handled `Block`s. In our original design, we had planned to create an `AbstractBlock` class and seven subclasses (each one being the type of block specified in the project guideline). However, we decided to allow more generality in the blocks allowed, so we instead merged all of the Block classes we had together into a single `Block` class. This way, we can simply specify blocks in the file `blocks.txt`. So, if we wanted to introduce a new type of block or remove a type of block, we can simply make changes to this file rather than making changes directly to the code.

As well, we originally planned for the `Game` class to handle making the blocks. However, this turned out to be a much larger aspect of the program than we anticipated. So, we opted to creating an entire other class to handle this, which is the `RandomBlock` class. This more clearly defined the role of the `Game` class, which now currently handles interaction between user input, the board, and the blocks.

The deadlines that were set up previously were not rigorously followed. This is because the difficulty and length of writing some classes was misjudged; the schedule shifted in a way that it became more weighted toward those classes. However, the general order of the development was quite similar to what was planned. More basic classes and methods that did not require other classes were written first, such as `main` and `Cell`. However, as we added new classes, we constantly realized that the classes we had previously written were not perfect and had to be changed according to the new classes. These changes varied largely, such as from including a pointer, to completely rewriting a fundamental method.

Questions

1. What lesson did this project teach you about developing software in teams?

When it comes to developing software in teams, the most important element is communication. If there is strong communication between the partners, then everybody understands the problem and it can be tackled with full force. However, if there is miscommunication, or some members are missing information, this can often lead to the misinformed party to attempt to solve a different problem that may not be necessarily relevant to the issue at hand.

On reflection of our development, we realized that toward the end of the project, our communication skills became much stronger, likely due to becoming more familiar with each other's programming and problem solving styles.

Also, another large aspect of communication is documentation of changes to code. We used a file share system called GitHub to store revisions of the code because it showed clear differences between versions. GitHub also provided a field for revision summary on upload. This allowed for easier understanding of changes made by our partners over the course of the development of this project.

2. What would you have done differently if you had the chance?

The order in which we developed our code caused it to be quite hard to test until the majority of the code was complete. This caused us to be quite uneasy about the state of our programs until very late into the development. Fortunately for us, the code we wrote did not have too many major issues and were able to be fixed after we put everything together. However, if we had not been as careful as we were during development, or it was an even larger project, the order of our programming could have been extremely fatal. Instead, we should have created a special debug suite to custom to our stage in development that allowed us to test the components that we have written. This way, we would know if what we have is working and what was broken. This would have let us debug the code continuously throughout development and avoid any uneasiness about the program's correctness throughout development.