CS246 A5: Quadris Fall 2017

**INTRODUCTION**

For this last CS246 Fall 2017 assignment we had three different programs we could implement, Quadris, Watan and Sorcery. We chose to implement Quadris, a non-real-time version of Tetris, mainly because of three reasons:

* Compared to the other options, we were most familiar with the game and the logic behind it, as we all had played it before.
* We considered that having this previous experience and being more familiarized with the game would make us feel more comfortable implementing it.
* We also liked the idea of implementing Quadris, as it was really similar to Tetris, a game that when released, had a huge impact on our childhoods

During the selection we also considered the Sorcery game. Though some of us were familiar with similar games such as Hearthstone, there were many warnings and little details that could lead to big errors. Due to all these warnings, we thought that it would be more sensible to avoid Sorcery.

Finally, we were not that familiar with Watan or any similar kind of game. This could lead to misunderstanding the requirements as well as making testing very difficult, so we just discarded it.

**OVERVIEW**

For the final design, we have included some new classes, as well as added new methods and variables to the original classes to adapt them and be able to implement all the functionalities. However, the main design structure was preserved.

Our design consists of a board where the game takes place, this board is partitioned by cells, which in turn specify their state. This board can also contain different types of blocks, which can be any block derived from the virtual block class. It that can be rotated both clockwise and counterclockwise, moved left, right and down and dropped upon user’s request. Finally, the Level classes specify different difficulties in which the game can be played. Probabilities are involved in block generation dependent on level as well as the ability to take in a seed for random generation

The Levels and the Board classes employ the factory method pattern and the Cells and both Text Display and Graphical Display interact following the observer pattern. The cells behave as the subject meanwhile the other two classes observe and wait to be notified.

A more detailed description of all the classes used in this project and a brief description of their functionality is outlined below:

**Board**

Principal class of our design and all the others are built around it. It is formed by cells, which are the pieces forming the grid. The board also contains previously dropped blocks, the current block being manipulated as well as a preview of the next block to come on screen. Lastly, it will also keep track of the level it is currently in as well as the score/high score of the user.

*Functionality*:

* Check when a row is full using “isRowFull” and when needed remove it in “removeLine”.
* Add a block to the board or erase it. We use “addBlock” and “eraseBlock” respectively.
* General moving actions:
  + move: down “moveDown”
  + horizontal: “moveBlockHorizontally”.Positive/negative parameter determines Right/Left movement accordingly
  + drop: “dropBlock”
  + rotate: “rotateClockwise” and “rotateCounterclockwise”.
* Check if a move is legal.
* Add a 1x1 block every 5 moves without clearing any line, “addBlankBlock”.

**Cell**

Units that form the board where the game will be taking place. Each cell will have a position of type Pos, and a state that will determine whether they are free or occupied by a specific block. It is also the subject of our observer pattern, so it can attach and notify observers.

*Functionality*:

* Attach and notify observers. “attach” and “notifyObservers”
* Update the state of the cell.
* (in cell we have a pointer to a block)

**Pos**

Convenient way of publicly storing x and y coordinates for each cell of the board. It is implemented in Cell.h.

**Observer**

Abstract class that all the observers of our board will have to derive from.

**Text Display**

Cell observer responsible for printing to console the current state of the board.

*Functionality*:

* Be notified and get updated when a cell is modified, “notify”
* Overload operator<< so that we get the desired output.

**Graphic Display**

Cell observer responsible for graphically displaying the current state of the board.

*Functionality*:

* Be notified and get updated when a cell is modified, “notify”
* TODO

**Window**

Needs a description TODO.

**Commands**

Input received by the user is interpreted and processed.

*Functionality*:

* Read the command with its multiplier, “readCommand”
* If sequence command received, switch input from user to given file. Once file is read, give control back to user
* Has a vector of strings where it stores all the possible commands, “masterCmdList”
* Interpret command shortcuts, “autofill”

**Global**

Storage of all the global variables needed in our program

**Block**

Abstract class from where all the specific types of block are derived. Each block creates himself and implements its rotate methods.

*Functionality*:

* Rotation works as the pure virtual methods that all concrete blocks will have to implement. “roateClockwise” and “roatteCounterclockwise”
* Implements moving a block down and horizontally. “moveDown” “moveHorizontally”
* Delete a position of a block when it disappears while removing a line. “removePosition”

**Level**

Abstract class from where all the different levels will be derived. Handles the creation of the next block depending on the current level of the board.

*Functionality*:

* Calculate and create next block in the game. “createBlock”
* Dependent on the level; probabilities specified for block creation as per assignment

TODO: \*I should somewhere mention heavy, Hamza implemented this in a different class. Each level has a getLevel method that identifies which level the block is in

**UPDATED UML**

For the most part, our UML…. TODO: Insert UML

**DESIGN**

*The Factory method Pattern*

This pattern was really useful for the design of this program because this way we would only need to define the interface for creating an object and then let the subclasses decide which class to instantiate. We applied this pattern to Levels so that when you call the createBlock method on a specific level object, the appropriate block would be generated.

*The Observer Pattern*

We decided to include and implement this behavioural pattern in our design because this allows us to define a one to many dependency between objects, so that when one changes state, all the dependent ones are notified and automatically updated. This ensures the grid is always updated.

The generic implementation of the observer pattern requires having both “attach” and “notify observers” methods in the subject class and a “notify” method in the Observers that will update the state when called. In our particular case Cell will be our Subject and both Text and Graphic Display will be its observers. Cell knows its observers as we have attached them to it, when the state of a cell changes the observers will be notified. If we only want to work in text mode or graphic mode then only the specified observer will will be notified.(Not sure if it works like this or I just made it up(?)) When notify is called we will update the grid at that position to its new state.

*Inheritance*

We used inheritance in our program mainly in two cases, Blocks and Levels. In blocks we decided to use inheritance because there was a significant amount of code that is common amongst all blocks; inheritance would avoid duplicate code. For example, all block children have identical “moveDown” and “moveHorizonatally” functionality. Thus, it makes sense to only code those movements in the base class.

Inheritance also makes the code much more flexible.If the return type of a method is the superclass or a pointer to it, then all the subclasses will be valid options.

In Levels, all block creation follows the same pattern; a block is created and then returned. It would be senseless and tedious to implement each block return type in each level class and as such we simply implement returnXBlock in the superclass.

*‘Owns a’ relationships*

We decided to use “owns a” relationship between out board class and both cells and block classes because we considered that destroying the board should also destroy the cells and blocks.

**RESILIENCE TO CHANGE**

Due to the use of design patterns in our design we managed to make pretty flexible implementation. This way we can easily add new features or modify partially some requirements without having to make a drastic change to our code.

*The block implementation*

We decided to make a base class block that will apply to all the different concrete blocks. To do this we looked for the feature that all blocks had in common but that had a different implementation for each type, so a pure virtual method to be able to generate inheritance. This was their capacity to be rotated and to create each type of block we will need to define the positions we want it to take, in other words its shape and the new state its going to represent. By implementing the rotation methods clockwise and counterclockwise we will have created a concrete block. This way each time we want to add a new type of block, we will just need to create a new class that inherits from block and implements the previously described requirements and add it to the state list. //Maybe some more (little) things.

*Commands*

Commands are handled in an “else if” chain in main. Adding a new command is fairly simple and modular. One would simply need to add the command to the master\_cmd vector for autofill functionality. The second, and final, step would be to add the corresponding “else if” statement where they can insert the desired behaviour of the newly implemented command.

\* Note that we exclude the discussion of the actual command implementation in the core classes as that is completely dependent on which command you’d like to support.

*Factory Method Pattern*

Level follows the factory design pattern which allows easy adding on of new levels. One would add the new Level\_X class (as a child of the Level base class) and create their blocks however they choose. This modularity is provided by the virtual Level superclass.

*Observer Pattern*

The observer pattern through its attach and notify observers methods allows modular behaviour in the context of adding on new observers. For instance, if one wishes to add HTML support for this game for a web version, they can attach themselves as an observer.

**ANSWERS TO ASSIGNMENT QUESTIONS**

*Question 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? Could the generation of such blocks be easily confined to more advanced levels?*

The approach we thought would be more appropriate to solve this requirement is to add a private member to all blocks. It would work as a counter; before a new turn starts, we add one and check if it is equal to 10. If so, it means that it has been in the board for 10 turns already and we just delete it and update the grid accordingly.

With this solution, it would be trivial to generate the blocks in any level as the counter is initialized in the block constructor.

*Question 2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?*

A solution we found to this problem was having an abstract class, Level, and all the other levels build from this base class. This design would allow us to be able to add new levels in a very simple way, namely by following the factory method. Each Level subclass must simply concretely implement the pure virtual methods in Level base class such as “addBlock”. Regarding the compilation, the new Level subclasses we will create will never affect the already existing ones, so when recompiling our program, we will only need to compile the new level added and not recompile the entire code.

Generic techniques which are simply best-practices in code should also be done, such as using forward declarations rather than unnecessary include statements. Including header files should only be used when the actual implementation in the included files is required. This way we will be avoiding unnecessary compilation dependencies. Another way of reducing dependencies and therefore re-compilations, would be to take all private members out into another class and replace them with a pointer to the implementation. This way we will not need to re-compile when the private members change.

*Question 3: 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? 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)? 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.*

Having a file such as “commands.h” that is included in main.cc can make this possible. This commands file can handle all user-input and commands, and do the appropriate commands. Since Level/Block are abstract classes, new commands could be implemented all the way down to each block of each level, just by implementing new concrete subclasses.

Similarly, renaming an existing command could be done fairly easily as well, in the commands.h file. If the “rename” command is passed in by the user, the variable name which stores the command such as “counterclockwise” can simply be changed. This implementation will require that the original commands are stored in variables as strings (for example).

Also, because of the fact that commands are in main.cc or in the commands file, and not coupled with any actual implementation of the program, it will be easy to change commands or rename them without overly affecting the actual board or levels. Furthermore, these variable strings of command names can be stored in an initial array/vector, named “defaultCommands” for example, and then a “macro” language can be implemented by simply calling different commands in the array/vector in a specified order.

TODO: Do we have any new answers to questions or modify them? Or we stick to the ones we had before? Also, remove question part? (TA knows Q’s)

**EXTRA FUNCTIONALITY**

For this, TODO:

**FINAL QUESTIONS**

*Question 1: What lessons did this project teach you about developing software in teams?*

For 2 of our 3 members, this was the first time using version control (Git) software which was an excellent learning experience as to how software development is conducted in industry. The most significant lesson we learned as a team working together on one major deliverable is the importance of communication of not only which member is tackling which task, but also on the requirements and behaviours of produced functions. Since all the functions in the program coexist and interact together, it is paramount that we define expectations for each function. For example, a team member working on a rotate method must communicate what parameters they require when their function is called by, perhaps, another team member.

Another important lesson we learned was the importance of only pushing code that compiles. Since there are many components to our final project, each being worked on by different people, it makes testing near impossible if others are pushing code that does not compile. If we enforce this rule amongst ourselves, it makes implementing new features and thus building the program much easier.

*Question 2: What would you have done different if you had the chance to start over?*

Given the chance to start over, our group would have done the following different:

* Divide work more sensibly: The way we divided work for this project was primarily based on how many classes each person was to implement. If we had the chance to do it over again, we would attempt to divide workload by logical flow of the program. By doing so, we would have less confusion of whose function provides/requires a function created by another user. It would allow for a more fluid design
* Started command skeleton earlier on: Our group experienced some challenges implementing the user interface system which required going into previously-finished classes to make modifications. Had we done all the command handling first, it would have been easier to structure the following methods/functions accordingly.

**CONCLUSION**

This last CS assignment was a great learning experience for all of us. We got to experience what collaborating towards a single deliverable was like all while employing various design strategies we learned in class. In particular, it was very beneficial for us that we used industry-standard version control for the development of this project as that is something we can directly carry outside the classroom. Debugging the program was also much easier as having a group of 3 allowed for more angles of attack and different ideas. We are quite proud of our work and hope you enjoy running it as much as we enjoyed building it.