Features Implemented

We have implemented all the functionalities specified in the quadris.pdf assignment page. In addition, we incorporated some bonus features.

The following additional commands are supported:

- **rename oldname newname** changes the name of the command from oldname to newname. It has no effect if oldname is not an existing command or newname already exists as a command

- **hold** holds the current block so that it can be used later. If a block is already held, the held block becomes the current block. After this command is issued, the drop command must be issued before this command has any effect again.

- **skip** skips the current block if it is skippable.

The following additional options are supported on the command line:

- **-rename** allows users to modify the names of commands using the rename command

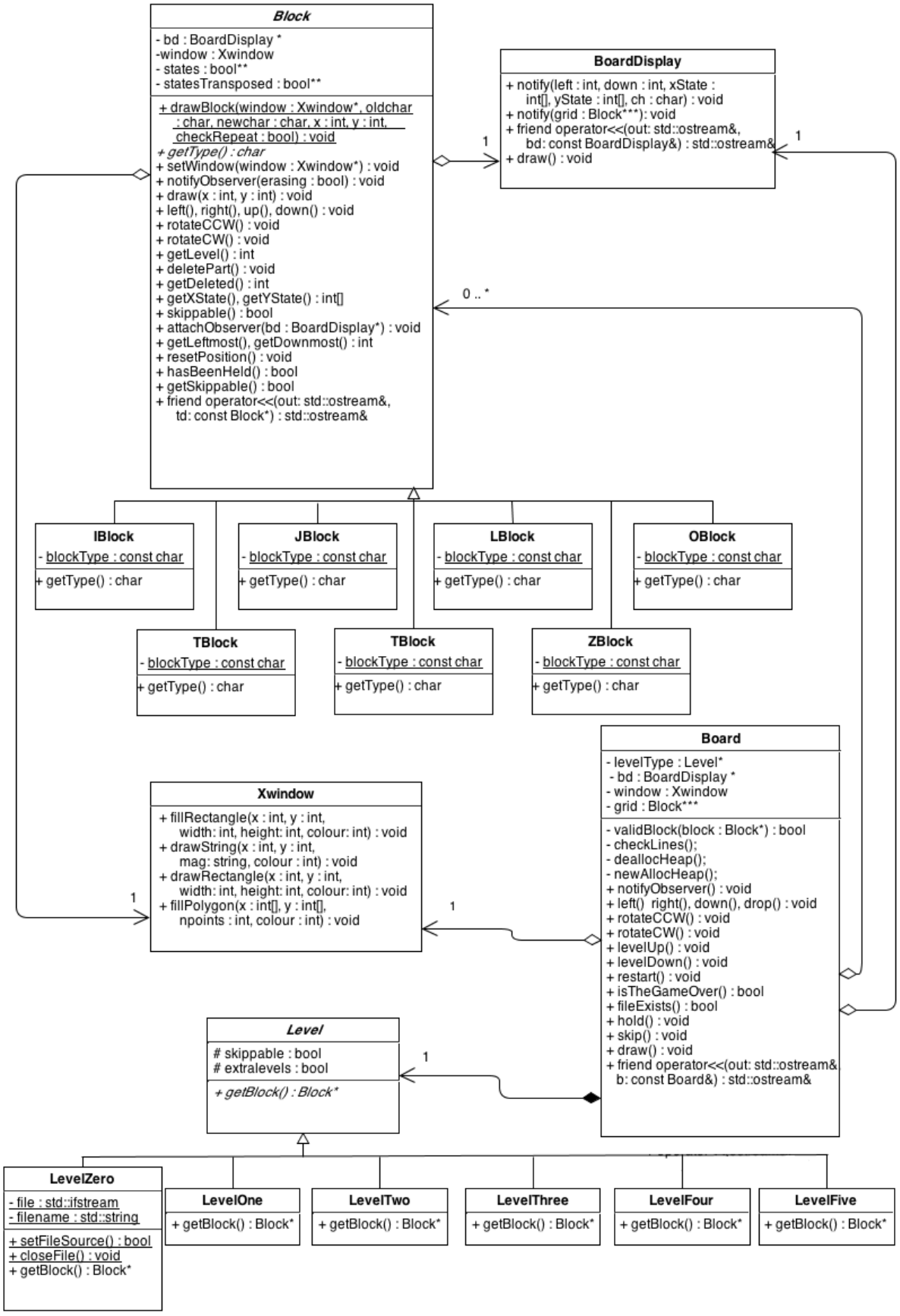
- **-extralevels** allows the game to be played with 6 levels instead of 4. Level 0 remains the same as the non-bonus program. Level 1, level 2, and level 3 of the non-bonus program becomes level 2, level 3, and level 4 respectively. Level 1 now cannot generate S and Z blocks and generates all other block types with equal probability. Level 5 generates S, Z, and T blocks with equal probability and cannot generate any other block.

- **-hold** allows blocks to be held using the hold command

- **-skip** allows blocks to be skipped using the skip command. Skippable blocks are generated at random. Without the -extralevels argument specified, the probability that a block will be skippable in level 0, 1, 2, and 3 are $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{5}$, and $\frac{1}{5}$ respectively. With the –extralevels argument specified, the probability that a block will be skippable in level 0, 1, 2, 3, 4, and 5 are $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, $\frac{1}{5}$, $\frac{1}{5}$, and $\frac{1}{4}$ respectively.

Note: By default, none of the behaviour specified by the above command line options is applied.

In addition, we have modified the Xwindow class to make the blocks appear 3-D on the window. We accomplished this by drawing the border of the block using trapezoids that are of a different shade of colour than the rest of the block.


Block
- bd : BoardDisplay *
-window : Xwindow
- states : bool**
- statesTransposed : bool**
+ drawBlock(window : Xwindow*, oldchar : char, newchar : char, x : int, y : int, checkRepeat : bool) : void
+ getType() : char
+ setWindow(window : Xwindow*) : void
+ notifyObserver(erasing : bool) : void
+ draw(x : int, y : int) : void
+ left(), right(), up(), down() : void
+ rotateCCW() : void
+ rotateCW() : void
+ getLevel() : int
+ deletePart() : void
+ getDeleted() : int
+ getXState(), getYState() : int[]
+ skippable() : bool
+ attachObserver(bd : BoardDisplay*) : void
+ getLeftmost(), getDownmost() : int
+ resetPosition() : void
+ hasBeenHeld() : bool
+ getSkippable() : bool
+ friend operator<<(out: std::ostream&, td: const Block*) : std::ostream&
BoardDisplay
+ notify(left : int, down : int, xState : int[], yState : int[], ch : char) : void
+ notify(grid : Block***): void
+ friend operator<<(out: std::ostream&, bd: const BoardDisplay&) : std::ostream&
+ draw() : void
1
1
0 .. *
IBlock
- blockType : const char
+ getType() : char
JBlock
- blockType : const char
+ getType() : char
LBlock
- blockType : const char
+ getType() : char
OBlock
- blockType : const char
+ getType() : char
TBlock
- blockType : const char
+ getType() : char
TBlock
- blockType : const char
+ getType() : char
ZBlock
- blockType : const char
+ getType() : char
Xwindow
+ fillRectangle(x : int, y : int, width: int, height: int, colour: int) : void
+ drawString(x : int, y : int, mag: string, colour : int) : void
+ drawRectangle(x : int, y : int, width: int, height: int, colour: int) : void
+ fillPolygon(x : int[], y : int[], npoints : int, colour : int) : void
1
1
Board
- levelType : Level*
- bd : BoardDisplay *
- window : Xwindow
- grid : Block***
- validBlock(block : Block*) : bool
- checkLines();
- deallocHeap();
- newAllocHeap();
+ notifyObserver() : void
+ left() right(), down(), drop() : void
+ rotateCCW() : void
+ rotateCW() : void
+ levelUp() : void
+ levelDown() : void
+ restart() : void
+ isTheGameOver() : bool
+ fileExists() : bool
+ hold() : void
+ skip() : void
+ draw() : void
+ friend operator<<(out: std::ostream&, b: const Board&) : std::ostream&
Level
skippable : bool
extralevels : bool
+ getBlock() : Block*
1
LevelZero
- file : std::ifstream
- filename : std::string
+ setFileSource() : bool
+ closeFile() : void
+ getBlock() : Block*
LevelOne
+ getBlock() : Block*
LevelTwo
+ getBlock() : Block*
LevelThree
+ getBlock() : Block*
LevelFour
+ getBlock() : Block*
LevelFive
+ getBlock() : Block*

Program Design

The Block class is responsible for the movement, rotation, and properties (such as the level it's generated in and whether it's skippable or has been held) of a block. The Block class is abstract, and seven concrete classes inherit from it, representing the seven types of blocks in the game. The BoardDisplay class stores a 2-D array of characters representing the current board, and is responsible for printing the contents of the current board to standard output and to the graphical display. The Level class is responsible for generating blocks. The Level class is abstract, and concrete classes representing the various levels of the game inherit from it. The Xwindow class is responsible for graphical output. The Board class generates Block objects using the Level class, and calls the appropriate functions in the Block class to manipulate the current block. The Board class also stores a 2-D array of Block pointers that represent the blocks that have been dropped. Thus, the Board class is responsible for line clearing and scoring. Board's operator << prints the scoreboard, and calls BoardDisplay's operator << to print the current board and Block's operator << to print the next and hold blocks. Similarly, Board::draw draws the scoreboard to the graphical display, and calls BoardDisplay::draw to draw the current board and Block::draw to draw the next and hold blocks. The main program, quadris.cc, is responsible for getting input from the user and calling the appropriate methods in the Board class based on the input. After each command, the main program calls Board's operator << and draw method. The main program is also responsible for processing options specified by command line arguments.
We applied the **factory method** design pattern to generate new blocks. The abstract Level class contains an abstract method getBlock. Each concrete subclass of Level implements the getBlock method based on the characteristics of that level. A polymorphic pointer to a Level object is stored in Board. We also applied the **observer** pattern. BoardDisplay is an observer of Block. When the positions occupied by the current block are changed, Block notifies BoardDisplay of its new positions. In addition, BoardDisplay is also an observer of Board. When a block is dropped, Board notifies BoardDisplay the new contents of the board since lines may be cleared.

For example, a typical command (take "left" for example) is executed as follows:

- The command is parsed and Board::left() is called
- The board's observer BoardDisplay is notified of the position change (erase).
- Board::left() calls Block::left() and moves the block temporarily.
- Board::validBlock() is called to check the validity of the new position of the block
- If it is invalid, Block::right() is called to move back to original position.
- The board's observer BoardDisplay is notified of the final position, (redraw)

And a "drop" command is processed like this:

- The command is parsed and Board::drop() is called
- The board's observer BoardDisplay is notified of the position change (erase).
- The Block::down() and Board:: validBlock () is called in succession until a false is returned.
- Block::up() is called to move it back into the last valid position possible

- All of the current block's positions is permanently stored into the 2d array grid (of Block pointers)
- Board::checkLines() is called to check for filled lines, to deallocate completely-deleted Blocks and to compute scores
- Level::getBlock() is called to receive a new block from the factory.
- The board's observer BoardDisplay is notified of the final position, (redraw)

It is also worthwhile to note that BoardDisplay is highly coupled with board and block, with the reason being that it was designed to be an extension of the two classes, in which they can be drawn. The other classes in our code have fairly low coupling, and good cohesion, as they each serve an individual purpose and operate largely independently of each other's implementations.

UML Changes

There are several differences between our original UML and design and our final UML and design. We changed the name of the TextDisplay class to BoardDisplay because the BoardDisplay class prints to the graphical display in addition to standard output. We originally planned to call methods of Xwindow in different methods of Board similar to notifying TextDisplay, but realized that doing so would complicate the code and result in unnecessary redrawing. Variables that represent previous statuses are stored in Board and BoardDisplay to prevent unnecessary redrawing. Block's operator << and BoardDisplay::draw were added for easier output. We overloaded BoardDisplay::notify as we realized that BoardDisplay needs to observe Board when blocks are dropped and lines may be cleared. We made Block::notifyObserver public as we realized a Block should not notify BoardDisplay if it is moved to occupy invalid positions. The Board class checks for the validity of the new location (in bound and not previously occupied), and calls the reverse method (e.g, call left then right) if necessary. A parameter was added to Block::notifyObserver to distinguish drawing versus erasing. We also realized more getter methods in Block were needed for Board to check the validity of the location and notify BoardDisplay. We originally planned on having fields in Block representing the positions relative to the bottom left coordinate for each rotation state, and write assignment statements for them. In our final design, 2-D arrays of bool representing the rectangle enclosing the block were added. True represents that the location is part of the block and false represents it is not. When rotation occurs, they are updated using loops. We also realized the notifyObserver and draw method can be put in the base Block class to reduce duplication. In addition, methods were added to signify game over to quadris.cc (Board::istheGameOver), check for missing file (Board::fileExists), and open and close file in level 0 (LevelZero::setFileSource and LevelZero::closeFile). Some methods in our original UML (Level::getLevelNumber, Block::undraw, Board::up, and Board::undraw) were deemed unnecessary and removed. Some differences between the original UML were due to bonus features. Block::drawBlock is static, and was added to draw a single box on the window. Since the colouring of our blocks is complex, it prevents the duplication of large amounts of code. Xwindow::drawRectangle and Xwindow::fillPolygon were also added for fancier graphical output. Block::hasBeenHeld and Block::getSkippable were added for the skip and hold features. Additional subclasses of Level were also added.

Plan of Attack – (actual)

Work Breakdown	Actual Timeframe	Person Responsible
UML diagram	July 14-17	Group
Four assignment questions	July 14-18	Split
Plan of attack version 1	July 14-17	Group
TextDisplay class	July 17-19	Xinpeng Zhao
Main.cc – command interpreter.	July 17-19	Siming Qi
Block base class	July 17-21	Split
Seven Block-derived subclasses	July 17-21	Split
Board class (movements and rotations, data storage and management)	July 17-19	Siming Qi
Board class (drop, scoring, restarts)	July 17-19	Xinpeng Zhao
Level base class and Level-derived subclasses	July 17-19	Xinpeng Zhao
Main testing period	July 20-22	Group
Graphics Drawing on Xwindow	July 20-24	Split
Command line arguments	July 20-23	Siming Qi
Extra features / bonuses	July 21-24	Split
Completion of missing documentation	July 26-28	Split

The above chart is a good summary of what we have done and the timeline of the completion of the project. This details also how we divided the work, and is in chronological order based on our timeframe of completion.

Group refers to working on the task as a team, discussing as we go.
Split refers to working on the task separately, discussing before we start, and at the end to eliminate inconsistencies.

Although there is a main testing period listed, testing of individual parts of the program occurred as the program is being put together. The main testing period serves only to indicate the period of time that we focus almost exclusively on bug fixes and testing for correctness and intended behaviour.

Changes from the original plan of attack:

- We managed to finish the majority of the coding for the program to run on the 19th. This was an early finish. We ran into trouble trying to compile, taking longer than expected, and along with general bug fixes and modifying Block, resulting in a later completion of the Block subclasses.
- During the testing period, we also started adding in new features one by one such as command line arguments and graphics, as well as the bonus content, resulting in a schedule shift.
- The bonus features also took more time and effort than previously imagined, due to unforeseen complications and information gathering across the web about XWindow modifications.
- Overall, we followed the schedule nicely, and it was a very good guideline for us to have to work efficiently towards set goals, as well as to evaluate where we are in terms of completion.
- Some individual parts of the work ended up being split between us, due to a new idea by a group member that he wanted to try implementing. Ex: graphics and block base class portions.

Questions on the Project Guidelines

Question 1:

We learnt that it is important to divide large software into many classes and methods as this allows multiple people to effectively work on the software at the same time. It is important that all members of the team agree with and understand the design before starting to code. We also discovered it is important to communicate with each other how we programmed the parts we were assigned to, at least at a high level. This often makes it easier for the other team member to add additional features to the program. This communication can be accomplished through discussing with the other team member and documenting the code. Furthermore, we discovered that programming to an interface rather than an implementation makes programming in teams easier. Group members can begin programming as if a method of another class existed while other group members are writing that method. Furthermore, it means one member can change the implementation of the part they worked on without resulting in additional changes to parts other members worked on. For example, we originally stored fields in the Block class that represent the positions relative to the bottom left coordinate for each rotation state. However, we later stored 2-D arrays of bool representing the rectangle enclosing the block. If we relied on the implementation of the Block class when programming in the Board class, we would have changed the Board class when making this modification to the Block class. However, we relied on the fact that there exists the getYState and getXState method in Block, which returns the x and y coordinates of the block relative to the bottom left coordinate. As a result, we did not have to modify the Board class. We also discovered that it is very beneficial for group members to test aspects of the program that other group members wrote in addition to testing aspects they wrote themselves. This way, when one group member misinterprets a specification, other group members can catch the bug. The group member who misinterpreted the specification would not have realized the bug.

Question 2

We know that our code was on the longer side, with almost 700 lines in board.cc alone. We feel like it is very possible to reduce code length to make it more readable in general by eliminating repetition where possible. For example, the left, down, up, right functions have the roughly the same functionality: makes a move, checks the move, and notifies the observers. This could probably have been abstracted into a method that encompasses these similar behaviours. On the topic of abstraction, we also feel that our many subclasses (6 derived from Level, 1 for each level we had, and 7 from Block) could have also been further grouped together, because currently, the only differences between the subclasses of Block is the starting configuration, and the only difference between LevelOne to LevelFive are the probabilities of each block generation and the probability of skippable block generation.
As well, we feel that too much “extra” work was done in re-implementing things a different way. For example, we had to re-implement the handling of the Block objects when we re-read the specifications and uncovered a small detail we had overlooked. Also, we had to re-implement the rotate method from accessing a pre-defined array of block rotation configurations to creating a rotation algorithm. This calls for carefully examining the requirements before writing the actual code and also to carefully review the plan, to ensure that the implementation actually works.

In addition to this, due to the complex handling of the Block class (the way we implemented block deletion), we could also benefit from using the RAII idiom, in terms of simplifying and shortening code. There is a lot more that we can improve on, especially as this is a larger project, for example: using more relevant design patterns in appropriate areas, increasing efficiency, and designing for reduction of coupling, and high cohesion. There is also much we can add, in terms of extra features that make the program easier to use and look better.

Questions on the Project Specification

Question 1:

The system can easily be designed to make sure that only the seven specified types of blocks are generated by using an abstract Block class, from which the seven Block types represented as subclasses are derived. We will only represent the blocks using Block objects, and will employ the use of a factory method within Level and its subclasses. By only calling the factory method to create a Block in Board, we can guarantee that only these seven Block types are created. Within each Block type (subclass), we will specify its standard (default) configuration and have a working rotate method that transforms the standard configuration into one of its four rotated states (rotated CW or CCW). With this working, we can also store the bottom-left coordinate of the smallest rectangle containing the block to perform the other movements, with the location of the blocks in the configuration specified relative to that coordinate. This serves to guarantee that each configuration that we create is standard, and that each configuration after each command execution is also a valid configuration.

Question 2:

The system can be designed such that each Block object contains two new fields : skippable and isTransparent, which are specified by command line flags, passed into board, and passed into the level "factory," then finally passed into the Block's constructor when the factory method is called from Level. If the flags are set, there would be an additional probability check to determine the specific block's skippable or transparency status. Executing the command "skip" would first check the current Block's skippable flag. The block then may be deleted and a new block gets generated by the factory. Implementing transparent blocks would be a bit more complicated. We will make the assumption that transparent blocks have the property that at least three of its four blocks have to be in empty spaces on the board for it to be a valid position. Then, we can modify our isValid() method, that checks positional validity after an attempted move, to check for the transparent flag. Instead of making sure that after a move that all four pieces of the block are still in empty space, it would keep track of the number of empty positions the block now lays on and say that the position is valid if the number is 3 or more. Otherwise, the attempted block move is reversed, as usual. In this design, a block can be made to be both transparent and skippable, by passing both in as command line arguments, and making the states have independent probabilities of occurring if the flags are set. The two are independent, separate commands, and should not interfere with each other.

Question 3:

The factory method design pattern can be applied to handle the various levels of the game. There can be an abstract class (Level) that contains a virtual method that returns a block. Concrete subclasses inherit from the abstract class, and implement the method that returns a block based on the characteristics of the particular level. The Board class, which controls most functions of the game, can contain a polymorphic pointer to a Level. The subclass type of the pointer will be determined by the current level of the game. To add additional levels to the game, new concrete subclasses that inherit form Level can be created. Those new subclasses will implement the method that returns a block based on the characteristics of the new levels. For the bonus, we did incorporate additional levels to the game.

Question 4:

The full names of each command as a string can be stored in a data structure, such as an array. The array can be traversed to match a command with the input from the user. If a command name starts with the input and all other command names do not start with the input, the input is interpreted as that command. The index number that corresponds to the command that the input should be interpreted as can be found. After the index number is found, the appropriate methods can be called based on the index number. To add an additional command, the array can be enlarged and the name of the new command can be added the end of the array. The change the name an existing command, the string stored in the array that corresponds to the command can be changed. To support a command where a user could rename existing commands, the array can be searched for the name that is to be changed (an input from the user). After the name is found, the content of the array at the index that stopped the search can be changed to the new name (also an input from the user). The name of the command that signifies renaming (for example rename) would also need to be added to the array that stores the commands. For the bonus, we incorporated this feature to the game. One detail that wasn't consider in the response for due date 1 was behaviour when the name of the new command already exists as another command. Changing the content of the array at the index that stopped the search first would cause unexpected behaviour in this case as only one command can be called now. We decided that the rename command should have no effect in such a case. We accomplished this by searching the array to determine if the new name already exists as a command.