Overview(overall structure):

Main function: parses the command and passes command line arguments to BiquadrisGame to start the game.

BiquadrisGame: Handles high-level implementation of the game, including stopping/starting, printing, communication with graphics, owns the two player boards and command manager.
Graphics: Renders graphics to X11 window.

Board: Owned by BiquadrisGame, handles logic of the board, including managing blocks, updating its own level, updating its own score, etc.

Block: The tetrimino that is manipulated on the board. Handles its own movement, rotation, etc.
CommandManager: Owns a list of commands. Handles calling commands, computing the minimum acceptable string for a command, etc.

Command: Abstract class for commands. Has pure virtual Call function.

LeftCommand, RightCommand, etc.: Concrete Command classes. Implement the Call function with their own behavior, taking a reference to the Board.

Level: Owned by a Board, handles the difficulty of a board and how new blocks are generated.
ScoreManager: Owned by a Board, handles the score of the board and updates the high score as necessary.

Graphics: Has an instance of a modified Xwindow class that was provided. Graphics renders the entire board, using various drawing functions defined in Xwindow.

Xwindow: Modified the color system to easily allow for custom hex colors, draw functions for different components such as the board, and made the screen utilize double-buffering to render the entire screen all at once during updates.

Updated UML Changes:

- Remove the Debuff decorator pattern classes. We found that at the time of implementing the special actions that they could easily be added in the necessary locations rather than creating a linked list of debuffs, since there was a lot of encapsulation in the BiquadrisGame class.
- Changed the coords property of Board to be a vector, rather than a fixed length, to support the asterisk block (only one square). Also allows for easier detection when a block is cleared.
- Renderer class was renamed to Graphics.

- For commands that take a file parameter in their Call function, this was removed to be identical to the parent class and instead they will read the input string themselves using the active input stream (either cin or from a file, if the sequence command was run).
- Many helper functions were added to Block and Board classes for encapsulation.

Design(implementations):

We used the Command pattern to implement the command system. This involved creating the CommandManager (Invoker) which has the ability to call commands using a virtual Call function. The commands all inherit from the Command base class and each implement their own functionality based on the type of command. For example, the DropCommand calls functions in the board that drop the current block and then generates the next block.
In order to run the Biquadris game, we used a loop to repeatedly take the next command from the input stream and then use the CommandManager to call that command on the board whose turn it currently is. There were some scenarios in which a command could terminate or restart the game. In these cases it was important to return from the command call before running any further logic in order to reduce recursion depth. For this reason we added a few flags to the game such as gameInProgress and turnInProgress which, if true, could prompt the program to restart or terminate.
Finally, we used unique_ptr and shared_ptr throughout the entire project so we did not have to write destructors for anything. We did not use raw pointers except in a few cases, for example Board has a pointer to its owning BiquadrisGame. This does not represent ownership so it does not need to be deleted when Board is deleted, and thus is acceptable within the extra credit requirements of the assignment.

Resilience To Change(support for changes in program spec):

Our Command system supports changes extremely well. It suffices to add a class which inherits from Command, and then add it to the command list in the CommandManager constructor. The CommandManager computes what the minimum accepted string should be for each command, so the developer does not need to worry about that.
Any graphics changes may be implemented in the Graphics::Render method. Graphics is well encapsulated so there are no other changes to be made. The two game boards are passed into the render method as well.
The addition of new Levels can be made through further logic in the Level::chooseBlock method. We chose not to use any rigid design patterns here for simplicity, since there are many stages at which the difficulty logic needs to run and it would overcomplicate things to use the strategy pattern or any other relevant design pattern.
The addition of new special actions can be made through the addition of a flag to the BiquadrisGame class. Our initial intention was to use the Decorator pattern for special actions, but due to the stage of implementation, it would not have worked well with the systems we had. The flags allow for the game to recognize which debuffs are active and apply them at any time.
In the graphics, there is support to add new colors by their hex color.

Previous Questions and how they differ now:

1. Question: 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?
    a. Previous Response: To achieve this, the Block objects can have a counter to track the number of blocks fallen. Once the threshold of 10 is reached, mark the block for deletion by setting its coordinates to -1, -1 in the next iteration. This action will prompt the Board to move down any blocks that were above it.

    b. Now: With our current implementation, we would instead loop through the blocks coordinates, going to each coordinate on the board and delete it from there and finally deleting it from the blocks vector, which would cause this block to be deleted since all shared pointers of it have been deleted.

2. Question: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?
    a. Previous Response: To accommodate the possibility of introducing a level with minimum recompilation would be to have a base abstract level class, which owns classes for each level. Each level object would have override a generation function to determine which piece would be selected. This would make adding new levels only require the compilation of new code of the levels. However, we have decided to only have a single level class with an if statement that determines how to determine which generation method to use. This helps simplify code structure.
    b. Now: Nothing different. Adding levels just like we mentioned before would be to add more statements to our if statement.

3. Question: How could you design your program to allow for multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?
    a. Previous Response: Implement the decorator pattern for effects. This allows adding all three blind heavy forces' effects to a general debuff object on the Board, avoiding extensive else-branches.
    b. Now: We decided to use a fixed number of boolean flags on board. It makes coding for effects a lot easier as it requires less boilerplate to create classes. We simply have multiple if statements that check for each, which then apply their effect separately. Adding more would require us to add more boolean flags to the board class, which is a very minimal amount of recompilation since it only requires compiling board.

4. Question: 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)? 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.

   a. Previous Response: Utilize the Command Manager, which manages command names and creates concrete command objects based on user input. This manager can handle renaming commands or creating aliases for existing commands, providing flexibility in command naming. To support a "macro" language, allow the Command Manager to interpret and execute command sequences defined by the player. This can be achieved by defining macros in the form of user-defined functions that execute specific sets of commands. Ensure the system can handle the additional shortcuts for macro commands effectively.
   b. Now: We still use the same command manager.

Extra Credit Features:

For extra credit features, we decided to code the project entirely with smart pointers for ownership. This was difficult getting used to, however with time it saved a great deal of time and thought into managing the memory life cycles of objects.

Final Questions:

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Developing software in teams requires a great amount of coordination. We spent a good amount of time discussing between each other and getting an opinion on how certain functionality should be executed and implemented. We also discussed and gave each other feedback on implementations and potential optimizations could be done.

In addition, we also communicated and coordinated quite a bit on completing certain portions of the project. Since we divided work into certain portions that all interlock with each other, we made sure to keep each other knowing of any changes and commits we made. This was to make sure we would be able to know when certain tasks that were blocked because they required something from the other person could be done.

Overall, this project taught us about the importance of communication and the coordination required of a team to effectively develop software.

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

We would have planned our use of design patterns earlier on. As mentioned before, our original intention was to use the Decorator pattern for the special actions, but since they were implemented late, the game flow did not play well with the way we were going to implement it. Instead, we should have planned our systems better around the special actions and integrated them from the beginning to properly execute the design pattern.

Additionally, we could have used the Observer pattern for graphics. This would allow for multiple windows and updates could be made at any time to the board which would immediately notify the graphics window. Our current method is to update once per action, along with the Print method which prints to the terminal. Using the Observer pattern would have provided greater flexibility with how we chose to update the graphics.