Final Design Document

Kefan Cao, Jacob Mausberg

August 13, 2021

1 Overview

Biquadris is packaged as a Game object. The main function constructs an instance of Game and then calls the playGame() method (playGame and constructor are the only two public methods of Game). Biquadris is implemented using the **MVC Design Pattern**. The Game acts as the controller, and is in charge of parsing commands and sending messages to the Boards. Each game owns two Boards. The Board is the model in the MVC. It stores a grid of Cells to keep track of the position of the Blocks. The Board is responsible for the creation of Blocks as well as Block movements (left, clockwise, drop, etc). To create Blocks, the Board has a Level factory, which uses the **Factory Method Design Pattern**. The Board also has a TextDisplay and a GraphicalDisplay. These are the views in the MVC. The displays are implemented using the **Observer Design Pattern**, where the displays are Observers to the Cells in the Board. In the next section, we will give a more in depth explanation of the implementation of these features.

2 Design

Command Parsing - The command parsing is handled entirely by the Game class' determineCmd() method. This diverges slightly from our DD1 plan, as we planned for this to be inside of a Controller class. We decided to merge Controller and Game into a single class to increase the simplicity and "cleanness" of our code. In the original plan, we wanted Controller to send commands to Game and then Game to send commands to the Board. This made the Game act as an unnecessary middle man, which would weaken encapsulation of the Game class (more public functions would be needed) and cause messier code. The implementation of determineCmd() is very similar to our initial plan. The Game stores a map of all of the Commands. The value is a Command enum that is used by the Game and the Board, and the key is a string that the user would input to trigger the command. The determineCmd() function reads in input and then parses it character by character. For each character, it compares it to all of the keys in the map, and removes commands that do not match. The process is completed when only a single potential match remains. This strategy allows for the user to input the minimum necessary command to be distinguished from other commands. The return value of determineCmd() is a Command-int pair, where the int specifies the multiplicity. This output is given to the Game's doCmd() method, which then sends the appropriate messages to the Board.

Coordinates - Coordinates is a very simple class we made that simply has an x and y coordinate. It is mostly used by the Blocks, but also by Cells, the Board, and the displays. It can be thought of more like a primitive type than a full on class. For this reason, and to keep things more readable, we did not include Coordinates in the final UML diagram.

Blocks - The Blocks are implemented as an abstract class, which spreads into the 8 concrete sub-classes for each block type (7 normal blocks + the brown block). A Block has a vector of Coordinates to store its position that it would occupy on the Board. Each sub-class has its own constructor to build the correctly shaped block (by filling the positions vector). Also each block implements the virtual getType() method,

which returns a character representing the type of the block. This is used by the displays. Additionally each Block stores the Level that created it so that the score for a cleared Block can be calculated correctly.

The abstract class implements all of the block movement functions (left/right, clockwise/counterclockwise, up/down). These functions simply alter the Coordinates in the Block's positions vector. The functions do not check if a move is valid based on the setup on the Board. The Board is responsible for checking, and uses the Block's getPositions() method. The validation process will be discussed more in the Board section, but note that it is the reason why a Block also has an up() method, even though this is not a command available to the players. Since the Board does the checking, the Block does not need to know any information of the Board at all, which minimizes coupling.

We will now explain how the movements work. The left/right and up/down functions just loop through the positions and alter the Coordinates appropriately, but clockwise/counterclockwise are a little more interesting. For clockwise, we first find the bottom left most square of the Block. Then we subtract this coordinate from all of the positions of the Block. This essentially moves the Block to the origin, so that the rotation formula will work. At this point we make the new X be the negative of the old Y and the new Y be the old X (this is the rotational matrix that we learned in linear algebra). Finally, we add back the bottom left to each of the positions in order to put the block back into place. For counterclockwise rotations, we just rotate clockwise 3 times.

The abstract class also controls the heavy status of a block. A Block has a private isHeavy bool. Heavy movements are not controlled by the Blocks, but rather by the Game. The Game communicates through the Board, which communicates with the current Block being dropped to see if it is heavy or to make it heavy. If the Game sees that the current block is heavy, then it tries to do an extra down motion. This is quite different from our DD1 plan for heavy. Over there, we said that we would use the Decorator Design Pattern and make Heavy a Block decorator. We were going to have the heavy decorator re-implement all of the movement functions to add a Down move after each movement. For example, a heavy Block's Right function would have first called component-¿right() and then component-¿down(). We ended up deciding against this idea for a number of reason. Firstly, we realized that it would not work to re-implement the movements like this. The problem is when you try do multiple movements at once, like "5right", it would cause a heavy Block to move 5 right and 5 down. But we wanted only a single down movement. Another difficulty was dealing with the requirement that when a heavy block cannot move down one, then it is dropped. So we decided to not re-implement all of the movement functions. Once we decided this, we realized it would be overkill to make an entire decorator class just so that a heavy block would return true when isHeavy() is called. Instead we decided to simply put the isHeavy bool inside of the abstract Block class, as explained above.

Cells - Cells are a wrapper class used by the Board to communicate with the Blocks and with the displays. Each Cell has a pointer to a Block and a Coordinate representing its position on the grid. It is through this class that we use the Observer Design Pattern. A Cell inherits from Subject, and its Observers can include a TextDisplay and GraphicalDisplay. Cell implements the Subject's notifyObservers() method. When a Cell notifies its Observers, it gives a reference of itself to the Observer. This reference gives the Observer all of the information necessary to update. Therefore, Cell has lots of useful data such as isBlind and boardNum. Most importantly, it has a getX() and getY() method to tell the Observer its position, as well as a getBlockType() method. The getBlockType() uses the Block's getType() method and returns a char. Also, it will return '?' if it is blind and ' ' it it is not occupied by a Block.

Board - Board is where the majority of the complicated logic involving Block movement and row clearing occurs. Board has a grid of Cells, which stores the actual setup of the game being played, as well as a vector of pointers to all of the Blocks that are currently on the Board. The Board receives instructions from the Game, which can call the following functions: dropBlock(), moveBlock(), functions to change the level, and functions for the special actions.

The moveBlock() function takes in a Command and multiplicity. There is a private function called moveBlockOnce() that does a single movement. moveBlock() calls this function multiplicity number of times or until it can no longer do the movement. Then after the moves are completed, the Cells are told to notify their Observers. The moveBlockOnce() function is tasked with only completing moves that are actually valid (i.e. stay on the grid and do not interfere with other Blocks). It does so by first doing the move, and then

checking. If check is unsuccessful then the inverse move is done. That is, if right() did not work, then left() is done to reset it. That is why the Blocks have both a down() and an up() method.

The dropBlock() function works by repeatedly moving Blocks down until it can no longer move. Once this happens, the grid is checked for full rows to clear. If a row is full then all of the Block pointers of the Cell's in the grid are shifted down by one. So for example, suppose we wanted to clear the third row from the bottom. Then any Cell in this row or above would have it's Block pointer switched to being the pointer of the Cell directly above it. Also, dropBlock() returns the number of rows cleared so that the Game can deal with this for special actions. If dropping the Block causes the player to lose, then the function returns -1. The reason that dropping could cause a game over is that when the block is dropped, a new current block is generated. We set current to be what was previously the next block and we call the factory make() function to make the new next block.

When rows are cleared, this is when the Board has to update its score field and send a message to the displays. Score keeping for a cleared row is rather simple, but score keeping for a cleared Block is more complicated. The Block clearing is done with the help of the Block vector mentioned earlier. The Board has a vector of smart pointers to all of the existing Blocks. Every time a Block is created (including the brown Block), it is added to this vector. As explained above, when a row is cleared, the Cell's block pointers get updated. By the time a Block is fully cleared, it should have no Cell's pointing at it anymore. Hence, the smart pointer to that Block has a count of 1, since the only pointer to the Block is the one in the Blocks vector. So to check for fully cleared Blocks, the program loops through the Blocks vector and checks if the count is 1. If the smart pointer reference count is indeed 1, (i.e. the Block should be cleared), then score increase is calculated using the Level that created the Block, and the cleared Block is erased from the Blocks vector.

The level changing functions, increaseLevel() and decreaseLevel() are quite simple. They just call the Level factory's increase() or decrease() functions and then send a message to update the displays. We will discuss these methods when we explain the Level class.

Lastly, we will explain how the Board deals with special actions. The Board has a makeCurrentHeavy() and isCurrentHeavy() functions for heavy, which controls the heavy status of the current Block. The Board also has a setBlind() function, which sets the blind status of the appropriate Cells. And the Board has a force() function, which makes a new Block of the inputted Block type, and sets it to the current Block, removing the previous current Block.

Level - Level is where new Blocks are made. The abstract Level class has a virtual make(), increase(), and decrease() function. These are all overridden by the 5 concrete Levels. The level's factory has a random engine as a private member. Upon construction of the concrete level factory, the random engine is seeded with a seed either provided by the player if the player wants to provide a seed as a command-line argument, or seeded by a random device. The make function of the level factory makes the block using an integer distribution that matches the Biquadris level specifications. Level 4 has a few extra functions that allows it to generate a brown block and check if it is time to make a brown block. The increase and decrease functions just return a pointer to the next or previous Level. So for example, the increase() function for Level3 would return a pointer to Level4.

Displays - The TextDisplay is an Observer of the Cells. The display stores a grid of characters, which get updated when notified by a Cell. The display has a method to print out a single one its lines. This allows the TextDisplays of each Board to be printed side by side. For the bonus feature, i.e, the textGUI, the TextDisplay has a function called Display() similar to the print function. The Display updates the textGUI with respect to the board's state. Additionally, the TextDisplay also has a function which enables the board class to update its score.

The GraphicalDisplay inherits from the Xwindow class (which was provided to us for the a4 bonus question). Like the TextDisplay, the GraphicalDisplay observers Cells and updates when it gets notified. A major difference compared to TextDisplay is that the Game is responsible for creating the GraphicalDisplay as opposed to the Board, which create the TextDisplay. This was not accounted for in our original UML but has been updated for the final version. The reason this is necessary is because both boards need to be displayed in the same window. Additionally, we only want a single Xwindow created for the entire duration of the program, so it makes most sense for the Game to create it and then to give a pointer of the display

to each of the Boards. Another complication with the GraphicalDisplay is that when the display is notified by a Cell, it needs to know which Board the Cell is coming from. Therefore, we added a boardNum() field to the Cells. Using the boardNum, and the Cell's Coordinates, the display is able to calculate the location of where it needs to update. Additionally, the GraphicalDisplay has methods to update the level, score, and high score that it displays. Lastly, when the "restart" command is given, the display whites out the whole screen and then redraws based on the setups of the newly created Boards. For colour in the graphical display, the provided colours in the window.cc file were not optimal for the tetris game, so we had to define our own colours to match the traditional tetris look.

Game - The Game class is where the entire Biquadris is packaged. Its only two public functions are its constructor and playGame(), which the main function uses to start the game. The Game is responsible for command parsing, as was discussed earlier. It has a doCmd() function that takes input from determineCmd() and then sends the appropriate messages to the Board. Game is also responsible for controlling which player's turn it is. The way this works is that the Game has a pointer to each Board and a pointer to the current Board (which is one of the 2 boards). The doCmd() function sends the instructions to the current Board, and also switches the current Board after each "drop" command is completed.

The Game is also responsible for prompting the user to input special actions when they are triggered, as well dealing with game over. There are two possibilities when a game ends: (1) both players have not yet lost or (2) one of the players has already lost. In the first situation, the winning player can continue playing in "single player mode" or may choose to restart the game. In the second situation, the player only has the option to restart the game. The way that single player mode works is that doCmd() never switches the current Board and special actions are disabled.

Additionally, the Game is responsible for the creation of the GraphicalDisplay as discussed earlier, as well as the creation of the 2 Boards. When a Game is created, it makes 2 Boards and the GraphicalDisplay. The GraphicalDispalay is never recreated, but only reset by drawing over it. However, when the "restart" command is given, the Game makes 2 new Boards, and the old Boards are automatically deleted.

3 Resilience to Change

Our implementation of Biquadris is very resilient to change. We created our classes using the single responsibility principle, and our classes all work together to create the game experience. The main classes that we have are the Game (controller), the Board (model), the Level factory, the Blocks, and the displays (view). Each of these modules are highly cohesive and minimally coupled with each other. The Game works with the Board and the displays and nothing else. The Levels are grouped together and mostly independent of other modules. Same goes for the Blocks. The Board is the most coupled with other classes, since it needs to make new Blocks, control Block movements, and send messages to the displays. But regardless of this, the Board does not need to know about the implementation of any of these modules since encapsulation is used.

All of this low coupling and encapsulation makes our program much more resilient to change. If anything needed changing, it would likely only affect a few of the modules. This allows us to make updates and fixes to our code with minimal recompilation.

The ways in which our design works to minimize coupling and maximize cohesion can be effectively seen in the UML. Some key examples of this are how Blocks and Levels are implemented. Both of these classes are built as an abstract class with sub classes extending out. The only class that knows about Levels and Blocks is the Board class and the Cell class, which is a middle man that the Board uses to communicate to the Blocks. Also, the Board only needs to know about the abstract classes but not the concrete Levels and Blocks. (In truth, the Board needs to know a little bit about Level3 and Level4 in order for the "random"/"norandom" commands to work and for making brown blocks). All of this minimizes coupling. It also means that there is high cohesion within the Blocks classes and within the Levels classes. They all work well together to package a highly usable and encapsulated product for the Board. This implementation

lends itself to a high resilience to change. New Block shapes can be easily added, or existing ones can be easily modified, with very minimal changes or recompilation needed. The same thing applies to Levels, as is discussed below in Question 2.

Additionally, since all of our modules are grouped together to perform a specific role (high cohesion), this makes it easier to add new modules and features to the game. Instead of redoing old classes, we could just make new classes that play new roles. Then since everything is encapsulated, we could more easily link these new features into the program.

Another major way in which our program is resilient to change is in command parsing. We explain this in more detail in question 4, but in summary, we can easily change the names of existing commands or add new commands with minimal recompilation. We could also relatively easily support things like command macros or allowing users to dynamically change the names of commands. We achieve this by storing a dictionary of commands and their associated names. Another reason why we can effectively do this, is since the name of the commands are only known by the Game class. The Game class "translates" commands to the Board, and the Board "translates" commands to classes like the displays and the Levels. This all minimizes coupling and increases resilience.

We will now explain some particular examples of possible changes to show how our software is flexible and resilient. One big example would be to add in more players or do single player mode. An advantage of our design is that each Board object is self contained (it does not need to know about the other Boards), and that the Game class is fully responsible for creating Boards. This means that to alter the number of players, we would just need to make changes to the Game class (as well as GraphicalDisplay, but this would be minor). Instead of the Game having a separate field for each of the 2 Boards, it could have a single vector that lists all of the Boards. And then when the TextDisplay gets printed, the Game loops through each line (how we currently do it), as well as each Board, and prints out a line. Also, instead of just switching the current Board to the other Board, switching the Board would set the current to the next Board in the vector. All of these changes would be fairly minimal and all would only require us to recompile the Game class.

Another big change we could accommodate is additional special actions. Our design is such that the Board has a separate method to trigger each special action. This makes it easy to apply multiple special actions at once (this is discussed in more detail in Question 3). As an example, suppose we wanted to add a special action that could fill in certain squares on the opponents Board (another interesting action could be to make certain squares empty). To do this, all we would need to change is the specialAction() method of Game and make another method in Board to do the effect, which we could call fill(). In specialAction() we would just add the command "fill" to the available commands to read in, and then when the user typed "fill", it would call the Board's fill() function. The fill() function would be pretty simple to make based on our design. The Board would need to store some "garbage" Block pointer to use to fill the Cells. Then fill() function would iterate through the Cells it wanted to fill and if the Cell is not already occupied, it would set the Cell's pointer to the garbage Block.

4 Answers to 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.*

There are two main ways that we could do this. Firstly we could use the **Decorator Design Pattern**. A block could be decorated with by a DisappearingBlock class that inherits from a BlockDecorator class. This decorator class would store the number of Blocks that have been dropped since it was created. The Block class would have a getCount() method, which would return the correct count for the decorator, but would return -1 for normal Blocks. The Block class would also have an increaseCount() method, which

would increase the count in the decorator, but do nothing for normal Blocks. Upper Levels could decorate Blocks with this feature after creating them, and the Board itself could decorate the current Block with this feature (similar to how the Board can make the current Block heavy). This approach was essentially our answer in the DD1 plan document.

A second approach would be to store the data directly inside of the base Blocks and not use the decorator pattern. This is more in line with how we actually ended up implementing the heavy feature. With this strategy, we would need to store the count inside of every Block as well as a bool to keep track of if it has the effect. The Block class would then have the additional methods of makeDisappearing() and getCount(). There would also be an increaseCount() method, which only increases the count of the Block if it is a disappearing Block (i.e. the bool is set to true).

In terms of updating the counts and clearing the blocks, in either of the above strategies, we would utilize the Board's blocks vector. This vector conveniently provides a list of all of the Blocks on the Board. We would loop through this vector and call increaseCount() on every Block pointer. While looping, we would also call getCount() to see if the count is 10, and if so we clear the Block. To clear the Block, we just loop through the Blocks positions and set the Cells in these positions to be pointing at the null pointer. That is the mechanism for how our program currently clears Blocks.

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

In this question, we will assume that the additional levels will be similar to the already existing levels (i.e. blocks are made with different probabilities and can have special effects), as opposed to adding in completely new game features. Otherwise, it would of course be difficult to predict what changes and recompilations would be needed.

In our design, the levels are implemented using the **Factory Method Design Pattern**. The Board has a pointer to an abstract Level class, which it uses to make new blocks. Each concrete Level inherits from the abstract Level class and overrides the make() method. As well, each concrete Level is made in it's own file. This means that adding a new level would just require implementing a new concrete Level class, which would need to be compiled.

The only other files that would need to be changed, are the levels right above and below the added level. This is due to how the increase() and decrease() methods work. For example, suppose we want to add Level5. Then Level4's increase() method would return a pointer to Level5, as opposed to doing nothing (currently, it does nothing since Level4 is the max level). This is slightly different from our initial plan, where we said that the Board would be responsible for increasing and decreasing the levels. We decided against this though, mainly to reduce coupling. In that original model, the Board would need to work with each individual Level, as opposed to just the abstract Level class. In our current implementation, the Board only needs to know about the abstract class (and a little bit about Level3 and Level4, but that is for a different reason), since the concrete Levels are themselves responsible for increasing and decreasing.

Question 3: *How could you design your program to allow for multiple effects to applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination.*

Each effect is applied independently of the other effects, since the Board has separate methods to apply each effect. As mentioned earlier, the Board has setBlind(), makeCurrentHeavy(), and force() functions, which are all fully independent of each other. Therefore, if we wanted to apply multiple effects simultaneously, we would just need to make a series of if statements (no long else-branches). For example, "if (a) do a(); if (b) do b();", allows us to apply any combination of a(), b(), both, or neither.

Applying multiple effects at once would be very easy for us to add to our current implementation. The Game's doCmd() and specialAction() methods are responsible for the special actions. In doCmd(), when a Block is dropped, we call the Board's dropBlock() method, which returns the number of rows cleared. Then doCmd() function checks if the count is at least 2, and if so calls the specialAction() function, which takes in the user inputted action and then applies it to the other player's Board. We could have just as easily called specialAction() multiple times instead of once. For example, we could have made it so that for every 2 rows

cleared, a special action is triggered (i.e. 4 cleared rows triggers 2 special actions).

Question 4: *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 name.*

As described earlier, our determineCmd() function reads in input and matches it to a command from a map of all the possible commands. This allows us to easily change command names or add new commands. To change a command name, we simply would change the key in the map that corresponds to the command who's name we want to change. The only thing we would need to recompile is the Game class. To add a new command, we would add the command to the map. We would also need to add a case in our doCmd() function to call this new command.

This implementation also makes it relatively easy to add a feature that let's the user change command names. We could add a "rename" command that displays the current dictionary, allows the user to select a certain command, and then enter its new name. The dictionary would then be altered, by changing the key for the command that the user selected to change, and everything else would continue to function as normal.

We could support the macro language feature by making a second map/dictionary, which stores the macros name as the key, and a list of commands as the value. The macro's name itself would also need to be added to the primary map of commands so that determineCmd() can find it. We would also make a doMacro() function, which takes in the list of commands associated with that macro, and loops over it, calling doCmd() on each command.

Alternatively, we could store the macro as a long string (as opposed to a list of commands), and using string streams make this the input. This is already partly implemented in order to facilitate the "sequence" command. Our Game stores an input file stream, and decides weather or not to read in from this stream or cin. Similarly, our Game could decide to read in from a string stream.

5 Extra Credit Features

For extra credits, we implemented a textGUI utilizing the ncurses library. For the textGUI, we added colours to letter blocks so they looked more like an actual GUI. Additionally, we enabled a feature which allows you to control blocks with your keyboard arrow keys. $\leftarrow$ for the command 'left', $\rightarrow$ for the command 'right' ect. Other commands have also been mapped according to the official tetris game. For example, space key for dropping, up arrow for clockwise rotation, 'z' for counterclockwise rotation, etc. This creates a pleasant user experience while playing the game, as the user does not have to type in words to move a block. Even with macros, it's a lot easier and pleasant to just press the right arrow five times if you wanted to move a block five spots to the right. Additionally, the textGUI, unlike the command-line text display, allows for updating instead of reprinting the screen every time.

Challenges were definitely encountered while making the feature. The biggest one may be that ncurses overwrites contents if the 'cursor' is on the same line, so we needed to make sure the display is printed in an appealing fashion and no additional characters that don't belong are printed. To do this, we kept track of our cursor position every time we update the screen to ensure that the screen is updated correctly. Another challenge that we encountered was that ncurses (textGUI) needed to have a way to enable special actions and needs to communicate this to the control method. To solve this, we added a method in the textGUI to request which special action the user chose and returned it to the control function. Since we did not want the request to the user to stick to the textGUI screen, we had to clear the lines, which meant we had to keep track of the lines that got printed along with the user input line. Finally, the last challenge we had was making sure that colours were similar to the tetris blocks. Colours in ncurses had to

be initiated in a central spot by their RGB values, which became tricky because the colours didn't want to display correctly, so we went with the alternative to use the preset colours, which were much more vibrant.

We also made two minor yet useful bonus features. Firstly, we made a -help flag to run the game with. When run with this flag, we print out all of the available flags that can be used to run the game. Secondly, we made a hint feature, as was very briefly alluded to in the instructions document. When the user enters "hint", we print out a list of the commands available to them. We do not print the commands such as "random" or "sequence" since these are meant for testing but not actual game play. These two features are not overly complicated to implement but are still very nice to have for first time users of the game.

Finally, we attempted and completed the memory management challenge. It should be noted that leaks will still occur if run with the GUI or ncurses, since xwindow and ncurses are known to have leaks. But if run with the "-text" flag, then there should not be any leaks. Here is the link which documents ncurses leak: `http://invisible-island.net/ncurses/ncurses.faq.html#config_leaks`. In order to complete the challenge, we were careful to strictly use smart pointers. It turned out that the smart pointers actually had some other useful benefits such as scoring for cleared Blocks.

6 Final Questions

Question 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?*

This project taught us many important lessons about developing software in teams and working in teams in general. For both of us, this was the largest software project that we have partaken in, as well as the first significant collaborative software project that we have done.

Working collaboratively brings with it many advantages, but also some new problems that must be considered. This includes how to effectively work on the same files, how to sync our edits together, and how to understand what the other person coded.

In terms of editing and working on the same files, we used git lab. This was our first time using git lab. It had a bit of a learning curve, but we now feel very comfortable using it. We learned about the different branches, about committing changes, and about merging the changes. We often encountered issues such as merge conflicts, but ultimately worked through them and eventually developed better plans as to avoid them. We realized that we should not be working on the exact same files at the same time in order to avoid conflicts. We always told each other about our merges, so that the other person could git pull, as to avoid conflicts later down the road.

Since we realized that we should not work on the same files at the exact same times, this brings us to another challenge/lesson. That is, time management and coordination of tasks. Since we were working as a team, we had to make schedules that worked for both of us. We had to find times to meet together or with instructors when we were both available. Also, we had to plan ahead and get certain code features completed by certain times in order to allow the other person to be able to work. We also learned how to delegate the tasks between us, so as to maximize our productivity and abilities to contribute. For example, we tried to delegate tasks that would take around the same amount of time to complete, and also matched tasks with the person that felt more comfortable with it.

Another important lesson we learned was in communicating and explaining our code with the other person. The communication element of team development makes clear comments in our code extra important. We had to make sure that our code was neat and easy to follow, so that both of us could understand it. Also, since we were just a two person team, we would often just call to explain our new additions and give any clarification for confusing code. This helped us practice the skill of explaining our design.

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

We feel very happy with the overall structure of our design. We spent a lot if time planning it out and then making refinements as we built the software. One thing that could be improved if we were to start over is that towards the end of the project, some of the implementation details of our code became slightly less planned out or purposeful. There were many small features that we had to consider, such as the "sequence" command for example. I think that this partly is due to the nature of the project, that we only had two weeks to build the software and that there were so many detailed specifications. However, if we were to start over, we could have payed even more attention to all of the smaller specifications, in order to be more mindful of them earlier on. Overall though, we are very happy with our design.

In addition to reflecting on our software design, we can also reflect on our work process and timeline. Looking back, it would have been a good idea to set up more meetings with instructors at the beginning of the project timeline, in order to clarify any questions we had about the instructions. This is especially true given how many specifications the software had. We ended up making more meetings with the instructors in the later stages of the project, which were very helpful.