The Plan

Our plan is to start from the basics and work from the ground upwards. The key implementation details of cell, board, block, and game are the most critical to get working, and the best way to be able to tell that our code is working as intended is to get textdisplay correctly outputting. The goal is that given a block, no matter how the block is generated, we're able to move it and have it interact with the board correctly, when moved / dropped. Additionally, cells should be correctly notifying the displays to ensure that what's happening internally is the same as what's being displayed. We should also be able to use the command interpreter with the most critical functionalities.

Afterwards, we will start implementing the features of the game, including rotations of the game pieces, special actions, and differing levels. With these come more complicated functions such as the random generation of blocks. The command interpreter should also be complete at this point, and we should be able to take in any of the specified commands, and play as close to a full game as possible. Scoring will also become important here, and we should make sure it works.

The plan is to have almost all of the work done far before the final deadline, which is why we have the least content needed in the third block of our time. In reality, this time will most likely be the most hectic, with time allotted for (most likely necessary) bug fixing. We will also be using this time to make a graphical display that works first, and then one that looks as nice as possible, as well as working on any enhancements if time permits.

As for the design document, we will be continually working on it as we implement features in our code. We will reserve some time at the end to answer the "final questions" as outlined in guidelines.pdf.

The distribution of our work and the deadlines we have set are as follows:

Deadline 1 - core gameplay + text display - Nov 28
Grace - Block internals and generation, relationship with Board
Ericsson - TextDisplay and observer/subject relationships, implementation of Cell
Jennifer - Game and Board, command interpreter

Deadline 2 - special actions / levels - Nov 30
Grace - Level 4
Ericsson - Force, Blind, Levels 1-2
Jennifer - Heavy, Level 3

Deadline 3 - final debugging + graphical display + answering final questions - Dec 2
Grace - Graphical display

Ericsson - Command QA
Jennifer - Display QA
Together - Final questions, enhancements, beautifying graphical display

Question 1
How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

We have a “level” field in our Board class, which keeps track of the current level. The handling of the differences of level are put into separate components: for example, the different probabilities when generating blocks for each level are handled in a single function and thus if we want to introduce additional levels into the game, we can simply add a section of code in that one function. Because our handling of level differences are segregated, only a single file or very few files need to be modified and thus we only require minimum recompilation.

Another possible approach is to have separate classes for each level and handle the differences through level objects, and we would only have to add new files or new classes when we wish to introduce new levels. However, we currently believe this is not the most clean or efficient approach and will stick to the first one.

Question 2
How could you design the 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?

In our Board class, we have fields that keep track of lasting effects (heavy and blind). To allow for a combination of effects, we can parse the user’s input word by word, and put each word through a series of if statements checking if the word matches an effect. By continuously reading effects in one at a time until input no longer matches, we can allow the program to simply loop on applying effects, with no need for an else-branch for different combinations.

If we were to add another effect, we can simply add another field that keeps track of it (if it’s a lasting effect and not just an action like force), and then add an if statement checking for it in our specialAction() function.

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? (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.

To accommodate the addition of new command names, we can store our commands in a map, to which we can add new key-value pairs. It would be easy to just add another key-value pair (where the key is the name of the command, and the pair is to be elaborated on below), and then match the user's command input by iterating through all the key-value pairs in the map as usual.

To support the renaming of existing commands, we can continue storing our commands in a map, where the key is the name of the command that the user enters, and the value keeps track of the original action of the command entered (before any renaming).

To support such a macro language, we can continue using a map, in which the keys are user-enterable commands and the values can be a string containing either one command, or a sequence of commands (delimited by whitespace). When the user enters a command, we find the corresponding value in the map, and parse it word by word, putting each word through a series of if statements checking if the word matches a command.