UML Diagram

See uml.pdf for the UML diagram.

Additional Notes

1. Result holds either a result value or an error value. It is a useful alternative to throwing exceptions.
2. For text input, the InputManager has a map mapping command strings to sequences of actions. The Level has its own map further mapping actions to sequences of actions.
3. The InputManager also has some hardcoded "keywords" which cannot be overridden and have special behaviours, namely, the "sequence" command. If we implement the extra features, we would also have keywords for define/undefine/rename.
4. Switching levels is done using the Level.pass_off() function, which returns a new level of the desired type with all data correctly transferred. The SessionConstants class contains a graph where vertices correspond to Level types and edges correspond to defined transitions (which are valid for pass-off). A path finding algorithm would determine an appropriate sequence of pass-offs to transition from a starting Level type to a desired Level type. It minimizes the amount of code needed to transition between levels, and minimizes levels' knowledge of other levels, since each level only needs to know its neighbours. Edges are chosen between similar levels to further simplify the transition code. It also gives more control to the levels, so if some levels store extra data, that data can be preserved when switching between them.
5. Levels also use the template design pattern with default implementations. Certain "trigger" functions like on_drop() and features like the action map allow for defining new behaviours with minimal changes.
6. Under the MVC design pattern, the InputManager and OutputManager collectively form the view, the InputManager is the controller, and the Session is the model.

Work Breakdown Structure

Colour highlighting: Komi Russell Richard
Time estimation is only there to indicate the approximate amount of work in the phase. We would move on as soon as one phase is complete.

Interface Phase 1 - 0.5 days

Complete by July 25, 2021

In this phase, we create all the files, includes, and empty class definitions. It's not a lot of work, so this is flexible.

1. Result

2. Action
3. Session, SessionConstants
4. InputManager
5. Level, Level0, Level1, Level2, Level3, Level4
6. Block, BlockCluster, BlockStyle, BlockClusterShape
7. DisplayManager, BCSSelector

Interface Phase 2 - 0.5 days

Complete by July 26, 2021

In this phase, we create all the function and method declarations, as well as expected data members.

1. Result
2. Action
3. Session, SessionConstants
4. InputManager
5. Level, Level0, Level1, Level2, Level3, Level4
6. Block, BlockCluster, BlockStyle, BlockClusterShape
7. DisplayManager, BCSSelector

Main Implementation Phase 1 - 2 days

Complete by July 28, 2021

In this phase (and the next) we implement everything.
We expect to interrupt this phase sometimes to resolve interface or integration related issues. However, we are not doing full integration yet.

1. Result
2. Action
3. InputManager (regular commands)
4. Level, Level0, Level1, Level2
5. Block, BlockCluster, BlockStyle (std::ostream), BlockClusterShape

Main Implementation Phase 2 - 2 days

Complete by July 30, 2021

The phase is split in case we find out halfway that we need to rebalance the work. These work items may ultimately be reassigned in order to rebalance the work.

1. Session, SessionConstants
2. InputManager ("sequence" keyword)
3. Level3, Level4
4. DisplayManager (std::ostream), BCSSelector

Integration and Test Phase - 1 day

Complete by July 31, 2021

At this point, we try to get a working executable. Each person is primarily responsible for the components they wrote, but anyone can poke into anyone's work to make necessary changes.

Extra Features Phase - 3 days

Complete by August 3, 2021

We are planning to potentially not have time for this.
Note that GUI is part of the core specification rather than an extra feature, but it is the largest feature, so we decided to schedule it for this phase. Better to have a working program first than to be too ambitious and not finish.

1. InputManager (other keywords, macros)
2. Session (graphical window and components)
3. InputManager (graphical inputs) - Komi
4. BlockStyle (gtkmm)
5. DisplayManager (gtkmm)

Design Questions

1.

How could you design your system to allow for some generated blocks to disappear from the board if they are not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?

The Block class would need a counter to keep track of how old it is. Additionally, the level class would need a change to its on_drop() function to remove these blocks once they are old enough.

This new feature would only occur on any Level subclasses that have it. It could be confined to more advanced levels by simply not adding this feature on the other levels, as all the existing levels do not have this feature.

2.

How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation? (Hint: a design pattern!!)

We would use the strategy pattern. Each level type is a subclass of the base Level class, and includes all behaviour specific to that level, as well as some behaviour which is common to all current levels but new levels may want to change.

The SessionConstants class includes a graph where each vertex is a level. The new level should be added to the graph, with at least 1 edge to connect it to the rest of the graph. The Session class knows how to find a path between the current level and the requested level, and perform the correct sequence of pass-offs.

For each new edge, code needs to be added on both ends to perform the appropriate pass-off. This means 1 or more existing levels will need an additional case in its pass_off() function.

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 the existing command names.

The InputManager already has a map between commands, which are strings that will be recognized from terminal input, and sequences of actions, which represent the various primitive actions a player can perform. Macros, renaming, and the like can be done just by changing the map. The InputManager handles the rest, including shortcuts.