CS 247 Project 2: Discussion Simran Thind, Janakitti Ratana-Rueangsri, Zuomiao Hu | July 16, 2021

B) Summary of UML

GenericBlock *Description: Represents the base class for the current block and next block. Each type of block will inherit from this class and will implement their own constructor to initialize the values of the following members accordingly.*

- **type_:** (I, Z, O, * etc.) Takes on an Enum type rep. the block type; will be used to know what characters/colours to use to display it
- **cells_:** A 4-long vector of 4-tuples each containing the relative coordinates (relative to bottom left) of the cells. Each index represents a rotation (i.e. cells[0] is 0 deg, cells[1] is 90 deg, etc).
  - For example, the S-Block would have the following cells:
    { [(0, 0), (1, 0), (1,1), (2, 1)], [(1, 0), (1, 1), (0,1), (0, 1)], [(0, 0), (1, 0), (1,1), (2, 1)], [(1, 0), (1, 1), (0,1), (0, 1)] }
  - We can use these relative coordinates and rotation to perform boundary and intersection checks from within the BoardModel.
- **rotation_:** Represents the index in cells_ for the current rotation.
- **x_, y_:** x and y are coordinates for the bottom left corner of the block in the 11x18 grid.

GenericLevel

- **boardModel_:** Pointer to BoardModel object.
- **generateNextBlock**(): Inherited Level classes implement how the next block is to be generated.
- **postMoveOperation**(): Called after every move in BoardModel (L. 0-2: do nothing, L. 3: call `down()` ("heavy"), L. 4: call `down()`)
- **postDropOperation**(): Gets called after every `drop()` in BoardModel (L. 0-3: do nothing, L. 4: If nonClearStreak is multiple of 5 - place block in middle (start from bottom middle and walk upwards until it finds an empty cell to put a 1x1 block)).

GameManager

- **isTextOnly_** / **seed_** / **scriptFile_** / **startLevel_** : command line args
- **controller_:** pointer to Controller object / **boardModel_**: Pointer to BoardModel.
- **textDisplay_**, **graphicalDisplay_**
- **start**(): Initializes the game. Starts an infinite game while loop that constantly reads input to the Controller. Game ends on reading EOF.

MVC Design Pattern classes:

Subject

- **observers_:** Set of observer views
- **subscribe**() / **unsubscribe**() / **notify**()

BoardModel derived from Subject *Description: Maintains current state of the playing grid .*

- **grid_:** 11x18 vector of chars representing cells. The char represents the Block type.
- **curBlock_:** Pointer to the Block that the player currently has control over
- **nextBlock_:** Pointer to the Block that comes next
- **score_**, **hi_score_**, **level_**
- **levelArray_:** Vector of pointers to GenericLevel object, each level at corresponding index (i.e. Level 0 at index 0)
- **nonClearStreak_:** Counts number of times the player dropped a block without clearing at least one row.
- **checkIfValidMove**(coords, rotation)**:** Is called by each of the "movement" functions below (i.e. left, right, etc.) before actually executing the move. Will use the curBlock's cells, rotation, and coordinates to see if any of the 4 coordinates overlaps with a filled cell on the grid.
- **checkCompletedRows**(), **calcScore**()
- **left**(multiplier), **right**(multiplier), **down**(multiplier), **clockwise**(multiplier), **counterclockwise**(multiplier): Calls `checkIfValidMove()` with the coordinates and rotation that the `curBlock_` would be at after the move. If it returned true, then actually update the `curBlock_'s` coords and rotation. Last, call `levelArray_[level_].postMoveOperation()`.
- **drop**(multiplier): Will read off the cells of the `curBlock_` and write them into the cells in the actual grid, and delete the `curBlock_` instance so that a new one can be created using the `nextBlock_`. Then, call `checkCompletedRows()`.
- **levelup**(multiplier), **leveldown**(multiplier), **norandom**(file), **random**(), **sequence**(file), **restart**(), **hint**()
- **I**(), **J**(), **L**(), **S**(), **Z**(), **O**(), **T**(): These will not call `postMoveOperation`.
- **loadGame**() / **saveGame**(): extra credit features for loading and saving game state to save.txt.
- **getGrid**(), **getCurBlock**(), **getNextBlock**(), **getScore**(), **getHiScore**(), **getLevel**(): Will be called by the Views to display the board.
- **getNonClearStreak**() / **setNonClearStreak**()

Observer

- **update()**

TextDisplay derived from Observer

- **boardModel_**
- **printHeader**() / **printGrid**() / **printNextBlock**()
- **update**() (override) Calls the BoardModel accessors and passes the data to the print functions.

GraphicalDisplay derived from Observer

- **boardModel_**
- **appObject** (Gtk::Application object, stored in a Glib::RefPtr smart pointer) / **appWindow** (Gtk::Window object)

- **update**() (override): Calls the BoardModel accessors and passes the data to the GTKMM graphics functions.

Controller *Description: Reads input, parses it, and calls the corresponding function in the BoardModel*

- **boardModel_:** Pointer to BoardModel.
- **commandList_**: vector of strings representing each command name
- **macroMap_**: Maps macro name to vector of each command in the macro.
- **operator**>>(): Pass `cin` input to `extractMultiplier()`.
- **extractMultiplier**(input) Extracts the number prefix, then calls `execCommand`, passing the input command and the multiplier
- **execCommand**(input, multiplier) Runs the input through an if statement where each case calls the proper BoardModel function
- **parse**(input, compareTo): Is used in each if condition to check if the input matches a command

C) Topical Program Flow

1. In **main()** -> Creates an instance of GameManager, passing in the CLI args.
2. In *GameManager*
    a. Create instance of BoardModel passing the following to the constructor (seed, scriptFile, startLevel).
    b. Create instance of appropriate Observer derived classes based on value of isTextOnly; subscribe them to the BoardModel.
    c. Create an instance of Controller passing BoardModel to the constructor.
    d. Begin an infinite while loop that reads user input from cin into the Controller input overload.
3. In *Controller*
    a. When Controller receives a user input via `operator>>`, it runs it through an if statement, calling the appropriate BoardModel functions.
    b. In each if-condition, it uses `parse()` on the input to pattern match (i.e. prefix matching).

D) Work Breakdown Structure

Task	Reasoning	Start Date	End Date	Assignee
Create header files and function signatures	We have already designed the structures and listed all the functions and signatures we need, this should be a small task.	Jul 15	Jul 16	Zuomiao
Implement Subject and Observer	These two classes are small (only about <5 functions and a few variables).	Jul 17	Jul 18	Simran
Implement Blocks	The blocks should be implemented before the BoardModel and GenericLevel, since they are used in both.	Jul 17	Jul 18	Janakitti
• Implement GenericBlock		Jul 17	Jul 18	Janakitti
• Implement all derived Blocks		Jul 17	Jul 18	Janakitti
Implement Levels	The levels are the fundamental features for this game. We want to get it implemented before the BoardModel. This should be doable within 3 days since the class is small (only 3 functions and not much differences between the derived classes).	Jul 17	Jul 19	Zuomiao
• Implement GenericLevel		Jul 17	Jul 19	Zuomiao
• Implement Level 0-4		Jul 17	Jul 19	Zuomiao
Implement BoardModel	This class plays a major role in the lifecycle of the game. We want to make sure this is completed at least a week before the due date to ensure we have enough time to integrate it with other components and test. The hint() function is saved for last and is assigned to all members since we expect it to require some more planning.	Jul 18	Jul 25	Janakitti
• Implement checkIfValidMove()		Jul 18	Jul 25	Janakitti
• Implement generateNextBlock(), postMoveOperation(), postDropOperation()		Jul 19	Jul 25	Janakitti
• Implement score calculation		Jul 20	Jul 30	Janakitti
• Implement all other functions		Jul 20	Jul 30	Janakitti
• hint()		Jul 20	Jul 30	All
Implement Controller	This is another major feature of the program. It also includes extra credit features. It would be best from the start to already support macros and renaming - hence why the length of time.	Jul 18	Jul 25	Simran
• Implement parse		Jul 18	Jul 25	Simran
• Implement macros + renaming **(extra credit features)**		Jul 18	Jul 25	Simran
Implement GameManager	This only involves one function to start the game using a while loop.	Jul 19	Jul 26	Zuomiao
Implement Game Saver in BoardModel **(extra credit feature)**	This is likely the most challenging extra credit feature, so we will work on it after the others.	Jul 20	Jul 26	All
Implement TextDisplay	This is the last feature to implement after all the main features.	Jul 19	Jul 30	Zuomiao
Implement GraphicalDisplay	Creating the GUI comes last because most if not all aspects of the game need to work first before they can be properly rendered. Setting up the env should be quick, but making the GUI will take some days.	Jul 26	Aug 1	Simran
• Set up environment for GUI testing		July 17	Jul 17	Simran
Create Makefile and make sure it works	We create a Makefile in the beginning and update it as necessary.	Jul 15	Aug 1	Zuomiao
Integration Test/Code cleanup	Ensure all components are in working condition by Aug 1 to dedicate time to creating tests and commenting code.	Aug 1	Aug 2	All

E) Design Question Responses Some responses refer to functions and models listed in Summary of UML.

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?**
   Assumptions:
   - We want to clear *squares* in the grid that were added 10 moves ago. This means that even partially-cleared Blocks should be cleared if any part of the tetromino exists for more than 10 moves.
   - Also, we do not need to worry about the scores since logically the player isn't responsible for clearing those disappearing blocks and they should not count towards the score.

   How we're implementing it:
   - The BoardModel can keep track of a timer int called `timer_` (can be mod 10 to prevent the int from going too high).
   - Each cell in the 11x18 grid in BoardModel will now store a pair<char, int>, instead of just a char. The char is the block type, and the int is the timestamp at which the cell was filled.
   - Each Block class derived from GenericBlock will have a `recordTimestamp_` boolean flag that indicates whether or not it should be deleted after 10 turns.
   - When the user places a Block that has the `recordTimestamp_` flag set to True, we write it's cells to the grid with the timestamp being the current value of `timer_`
     - If the Block does not have the flag, we simply put -1 as the timestamp.
   - Whenever the user drops a block, we will already be iterating through the grid to check for completed rows in `checkCompletedRows()`
     - While we do this, we can also check to see if the current cell we're looking at has a timestamp difference of 10 from `timer_` (i.e. equivalent mod 10), and delete the cell as necessary.

   How to confine to more advanced levels:
   - The concrete implementation of `generateNextBlock()` in the derived GenericLevel classes will be responsible for toggling the `recordTimestamp_` flag when it instantiates a GenericBlock derivative, based on the rules of the level.
     - For example, if we want Level 5 to include this disappearing block behaviour only for Z-blocks, we would implement `Level5::generateNextBlock()` to set the `recordTimestamp_` to true whenever it instantiates a ZBlock object, and false for all other blocks.
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!!)**
   How we're implementing it:
   - We employ the Strategy pattern to make it easy for future developers to develop new levels and tap into the various stages of the game lifecycle to perform level-specific operations. The Strategy pattern for levels allows the program to hot-swap levels during runtime.
   - The BoardModel has a vector of levels called `levelArray_`, that holds GenericLevel pointers. Each index in the vector represents a different level (i.e. index 0 holds the Level 0 derived class pointer). Each of these levels represents a different "strategy" for performing common game lifecycle operations: generating the next block, performing operations after a move, and performing operations after a drop.
     - The varying strategies are implemented through the use of polymorphism; different levels provide different concrete implementations of each of these three operations.
   - The current level is determined by which index the `level_` member of BoardModel points to in `levelArray_`. The program can easily change levels mid-game by changing the value of `level_`.
   - The addition of new levels only requires adding a new GenericLevel derived class and including it in the `levelArray_`, thus requiring little recompilation.
3. **A. 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 can store the command names in a vector of strings called `commandList_`. Each command described in the program specification will be assigned some index in this vector. In the if-statement block in `Controller::execCommand()`, each case will use the parse function to compare the user input against a command in the `commandList_`. For example, `if(parse(userInput, commandList_[0])){ // perform operation 0 ... }`
   - The reason why we use a vector instead of a map that maps human-readable identifiers to command names (for instance, instead of using array indexing, use something like `if(parse(userInput, commandList_['left'])){...}`) is to make the process of renaming commands easier.

   B. 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 we're implementing it:

- We can implement a new command: **rename**, that takes in two arguments - the first argument being the original name of a command and the second being the new name for the command. For example, in the terminal this could look like: `rename counterclockwise cc`.
- **rename** would also be stored in the command name vector of strings `commandList_`, and upon reaching the relevant If-statement case in the Controller, we'd first check the vector to see if we have a duplicate name (going back to the example, making sure "cc" - the second argument - doesn't exist in the vector), then we'd search the vector for the first argument, and then update that element.

C. 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.

Assumption:

- If Question C were to work in conjunction with Question B, macros that have the same name as commands are allowable, because we preface executing a macro with a **macro** command (**new, save,** and **exec** are parameters that cannot be changed - however **macro** itself can be renamed to something else as it would exist in command vector `commandList_`. That's why we also store command information in tuples (see below), so that if any command name changes we don't need to update information for macros (see below).

How we're implementing it:

- Have a map that maps macro names to a vector of tuples<int multiplier, int command_index, string args>, called `macroMap_`.
- Each tuple holds the: multiplier for a command (1 if it isn't supplied), the index representing the command in `commandList_`, and a single string representing the arguments to that command (can be empty if a command has no arguments).
- We store the *index* of the command as opposed to the actual name so that any modifications to the command names (from B) don't affect the macro.
- We implement three new commands for macros: **macro new**, **macro save**, and **macro exec.** For **macro new**, the first argument is the name for the macro, and the rest of the inputs are commands (with any relevant respective arguments) to be included in the macro in sequential order. Each command is separated by a newline. We end all this input with **macro save**.
    1. **`macro new`** `<name of macro>`
        - Initialize an empty vector, and store name of the macro in a temporary string variable.
    2. Keep reading in inputs for commands and store them in a vector (until **`macro save`** is inputted).
        - Examples of lines would be any command, like **`clockwise`** or **`norandom <file>`**
        - Create a tuple <int, int, string> (we'd split the input string of the line by after the first occurrence of a whole number, then first whitespace, so that the first subsection of string is the number multiplier, the second subsection is a string of the command to find in `commandList_` and get the index in `commandList_` from, and the last subsection is the string of args to store in the string part of the tuple).
        - On the temporary vector, push back the tuple.
    3. Store the vector in the map under the macro name key.
- To execute a saved macro, enter **macro exec <name of macro>**, and the program will lookup the name in the map and iterate through the vector to execute the commands
    - At each iteration, it would simply call `extractMultiplier(input)`, which in turn calls `execCommand(input, multiplier)` to execute the functions of BoardModel
    - Since the commands in the macro are stored in the 3-tuple format in `macroMap_`, we need to concatenate their values into a single string to pass as the `input` parameter of `extractMultiplier(input)`. This is how that will be done (note that `i` is from a for loop iterating through the vector of macro commands):

```
input = macros['macro_name'][i][0] +
        commandList_[macros['macro_name'][i][1]] + " " +
        macros['macro_name'][i][2]
```

F) Extra Credit Features

- **Macros / renaming commands**: See Design Questions question 3.
- **Smart pointers / minimizing explicit memory management**: Instead of using raw pointers, when we instantiate new objects and reference to objects we'll use unique and shared pointers as much as possible instead. All storage (including that mentioned in the design questions) use STL containers as opposed to more basic structures like arrays.
- **Save game to file**: This feature allows players to save the current state of the game to a file called `save.txt` and resume at a later time. This would require us to implement a **save** command that calls a `saveGame()` in BoardModel that saves a snapshot of all the members and encodes it in text format (in a format similar to JSON, with key-value pairs). We would also need to implement a **load** command that calls `loadGame()that` reads in the text from `save.txt`, and parses it to restore the values in BoardModel.
- **`enableBonus` CLI arg**: We can implement a flag that disables the extra credit features. For **Macros/renaming commands** and the **save game** features, the If-statement cases in `Controller::execCommand()` can use an `enableBonus` boolean in a logical conjunction with `parse()`. So even if the user inputs a command that matches one of these extra commands, it will not execute the command if `enableBonus` is false. For the smart pointers, this will not be toggleable.