Biquadris

CS246 Final Project

Overview of Plan

We plan to center our implementation of Biquadris around the following OOP principles that we have learned over the duration of this course:

1. "Program to interfaces, not implementations"
    a. The core behaviour is defined by abstract interfaces → Block, Level, Display & Special Action
    b. The client-side code will consist of pointers and references to interfaces → this allows concrete subclasses to be swappable (i.e. Player & GameEngine)
2. High cohesion, low coupling
    a. Each class has a well-defined goal: Grid will manage the state of the board, Player facilitates communication between Grid, Level and Block and the GameEngine is responsible for controlling turns and taking I/O.

To follow these principles, we will begin by building our core data structures (Player, Cell, Grid, Block and Level). Next, we plan to implement the methods that control our game logic. Then, we will build out the display components (TextDisplay, GraphicsDisplay, CommandInterpreter) and ensure that we both test our components in isolation and that they are well integrated. Our last step will be to implement any special features and polish our code (i.e. better documentation).

Timeline of Implementation

As a group of 2, we agreed to adhere to the following schedule to ensure that we produce the best quality of work in the given timeframe. The schedule below relies heavily on our current UML (which might be subject to change as we continue working on the project).

Step 1: Implement Core Model → Grid, Cell and Block **[2 Days]**

Group Member #1: Sharuga

- Implement Cell
    - Fields
    - Methods
- Implement Grid
    - Fields
    - Methods

Group Member #2: Tanisha

- Implement Abstract Block (superclass)
    - Fields
    - Methods
- Implement all Concrete Blocks (subclasses)
    - IBlock, JBlock, etc.

<u>Checkpoint:</u> Create a simple Grid and add a few Blocks, verify the move using isValid() and placeBlock() functionality.

<u>Possible Test Cases</u>

- Placing a Block in a full Grid
- Rotating a Block near a Wall
- Replacing cleared rows

Step 2: Implement Game Rules → Player, Level **[2 Days]**

Group Member #1: Sharuga

- Implement Level System
    - Note: ensure Grid has no direct access → it should only deal with block generation and level-specific rules

Group Member #2: Tanisha

- Implement Player System
    - Fields
    - Methods
        - Movement, Block management, Scoring, Controls, State

Checkpoint: Ensure all functionality works when a Player is created with a specific Level (verify score updates and level changes).

Possible Test Cases

- All levels generate the correct block types
- Level-up/down functionality
- Score updates when a row is cleared

Step 3: Implement GameEngine, CommandInterpreter **[1 Day]**

Group Member #1: Sharuga

- Implement CommandInterpreter
    - Fields
    - Methods

Group Member #2: Tanisha

- Implement GameEngine
    - Fields
    - Methods

Checkpoint: Run a basic game using only text; read commands from both CL and a file.

Possible Test Cases

- Invalid commands don't cause the program to crash
- Test command parsing (i.e. left, down)
- Test abbreviations

Step 4: Implement Display & Observer Pattern **[1 Day]**

Group Member #1: Sharuga

- Implement TextDisplay

Group Member #2: Tanisha

- Implement GraphicsDisplay → Could be a collaborative task

Checkpoint: Ensure the Grid correctly notifies all observers on every state change (movement, rotation, placement, row clear, level change). Verify that both displays stay perfectly in sync and never fall out of date.

Possible Test Cases

- Observers attach/detach as expected
- Display updates after every turn
- Display doesn't change when level changes

Step 5: Implement SpecialAction and Final Touches **[1 Day]**

Both Group Members:

- Implement SpecialAction subclasses
    - Blind, Heavy and Force
    - Integrate with the applyEffect player method (might be time-consuming)
- Testing
    - Create unit tests for Grid
    - Create unit tests for Level
    - Rigorous testing
- Clean up any code, provide clear documentation, reduce duplication and review discrepancies from previous UML

Checkpoint: Play a full turn where a player clears 2+ rows and triggers a special action. Confirm that blind, heavy, and force each apply correctly to the opponent and stop at the right time. Make sure multiple effects can exist together, and that the game still runs normally with movement, dropping, scoring, and display updates.

Possible Test Cases

- Blind, Heavy and Force features work as expected

Answers to Design Questions (based on our current UML)

---

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?

In order to support blocks that disappear after 10+ blocks have fallen, we can extend our Block class's functionality with fields such as timed(bool), lifespan(int) and spawnTurn(int). Higher levels would use the boolean flag to mark certain blocks as timed and set spawnTurn to the current turn count from Grid, with a lifespan of 10. Since our Grid implementation already tracks turns, we can modify Grid to check for expired timed blocks whenever a block is placed or rows are cleared. Only these higher levels have the ability to generate timed blocks, so this feature would not affect lower levels. Further, this allows us to successfully keep the generation logic inside Level, the removal logic inside Grid and ensures that GameEngine isn't involved in any of these details.

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

Since the game is designed to program to the Level interface, adding new levels will require minimal changes! Player only stores a unique pointer to a Level, and so, a new level means that we would have to implement a new subclass (i.e. Level5) and register it. Because all other components depend only on the abstract interface, the rest of the system remains untouched.

3. 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?

The SpecialAction interface is designed to support multiple simultaneous effects by allowing the Player to maintain multiple active effects simultaneously, represented either as separate flags or as a collection of active SpecialAction objects. Each concrete effect (blind, heavy, force) modifies the player's state with the apply() method; player actions refer to these states when moving blocks/rendering them. Adding a new effect would require the creation of a new subclass and hooking it into the point where effects are awarded. Since each effect is isolated and is communicated through methods in Player, the system is flexible, and no large conditional chains are needed.

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 the 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.

Our design handles all the commands within the CommandInterpreter, so changes to the commands will ideally never affect the rest of the game. The interpreter stores a table that maps user input (i.e. abbreviations) to the standard name our game engine uses. This makes renaming easy as we just have to update this table. The abbreviation problem can also be resolved by checking whether the inputted prefix matches exactly one command in our table. Adding new commands ⇒ adding an entry to this table and a matching handler in the GameEngine. Macros can also be handled in this way; the interpreter can store macro names and the list of commands they map to. From here, we can expand them during parsing before sending them to the engine, which only deals with the standard command names. The renaming, shortcut, macro logic is all contained within the interpreter, keeping the rest of the program simple and relatively unchanged!

Summary of Plan

In summary, our plan is to…

1. Build and test our core functionality first → Cell, Grid, Block
2. Add Player and Level to handle game rules and logic
3. Add GameEngine and CommandInterpreter to manage input, turns and higher levels
4. Implement our Display subclasses and observer pattern for both text and graphics
5. Finish with SpecialAction mechanics, advanced level features and more comprehensive testing

The responsibilities and estimated dates for completion of each subtask were clearly divided and agreed upon by group members.