CS 246 Group Project - Biquadris

Wendy Zhang (w78zhang)
Grace Yin (g6yin)
Chloe Chan (c79chan)

Tuesday, November 19	
Content	**Responsibilities**
● Finish UML's basic structure ● Set up GitHub repository ● Set up basic files and the dependencies between them ● Divide up writing parts for Plan of Attack	● All members
Wednesday, November 20	
Content	**Responsibilities**
● Work on Plan of Attack ● Think about various UML functions ● Think about private methods and subjects ● Think about return type and which patterns to use ● Finish questions	● All members
Thursday, November 21	
Content	**Responsibilities**
● Meet at Grace and Wendy's place at 3 PM ● Finish Plan of Attack ● Finish UML ● Finalize schedule ● Submit Plan of Attack	● All members
Friday, November 22	
Content	**Responsibilities**

● Start working on abstraction:
  ○ studio/main
  ○ block
  ○ level
● Start working on creating a structure for main
● Create reading input and output streams
● Create players and the functions within
● Create Level 0
● Create block abstraction and 1 block
	● *Grace*: studio/main
	● *Wendy:* block
	● *Chloe*: level

Saturday, November 23	
Content	**Responsibilities**
• Meet up and discuss how to format things at Grace and Wendy's place after midterm • Work on text, then graphics • Finish Level 0 • Complete studio code • Create the other blocks and next levels • The blocks need to have structure to store coordinates • 3 blocks should be completed • Make sure clear and restart work	• *Grace*: put in structure for main • *Wendy*: implement abstract block and at least 3 blocks and their up and down • *Chloe*: finish abstract Level and Level 0
Sunday, November 24	
Content	**Responsibilities**
• Meet at Grace and Wendy's place to continue coding at 4PM • Complete the rest of Block • Adjust main function to fit additional parameters • Test game engine • Think about edge cases • Make print for text only • Make sure score works for high score and low score • Create test cases	• *Grace*: finishing up main function • *Wendy*: complete Block and debug • *Chloe*: test cases and implementing more levels (Level 1, 2)
Monday, November 25	
Content	**Responsibilities**
• Complete graphics and ensure that it works • Ensure basic requirements (Level 0, left, right, down, drop, score, blocks showing up properly) are completed and work • `operator<<` should be working for overloaded function to print Board, Block, and Level • **`clearLine()`** and **`restart()`** must also be completed and working	• All team members will be testing and debugging the basic demo
Tuesday, November 26	
Content	**Responsibilities**
• Ensure all rotation blocks work • Begin writing final report (introduction and	• *Grace*: graphics • *Wendy*: rotation (**`rotateCC()`** and

structure) ● Implement `rotateCC()` and `rotateC()` ● Continue to enhance graphics with corresponding changes ● Complete remaining levels and implement heavy	`rotateC()`) ● *Chloe*: complete levels 3, 4 and write final report
Wednesday, November 27	
Content	**Responsibilities**
● All levels should work ● Continue working on final report (questions, design)	● Each group member will take on a level to test and debug ● *Chloe*: work on report
Thursday, November 28	
Content	**Responsibilities**
● Implement special actions (if all goes well) ● Implement graphics ● Continue working on final report (resilience to change, final questions)	● *Grace*: implement graphics ● *Wendy*: implement special action ● *Chloe*: finish up final report
Friday, November 29	
Content	**Responsibilities**
● Implement bonus mark features ● Edit final report and add in conclusion ● Submit the assignment 2 hours (at 3PM) before the deadline to ensure there is a submission	● Team will implement bonus marks, controlling with left/right controllers ● *Chloe*: complete final report

Summarization of Our Plan

Our plan for implementing Biquadris begins with creating a playable demo for `Level` 0, incorporating essential commands such as down, drop, left, and right to establish the foundation for basic gameplay.

From this starting point, we will progressively enhance the game through four well-defined stages. In the first stage, we will develop a functional prototype with text-based output to establish the game's core structure and mechanics. The second stage introduces rotation mechanics for the game pieces, leveraging the foundational architecture to ensure smooth integration. The third stage will add graphical elements and special actions, a step that may require significant architectural adjustments to support enhanced visuals and interactivity. Finally, to ensure stability, we aim to freeze major changes by Thursday, dedicating Friday to thorough testing, fine-tuning, and exploring optional bonus features. This phased approach

ensures we maintain a functional project throughout the development process while leaving room for complexity and refinement.

Our team has distributed responsibilities based on individual expertise to maximize efficiency. Chloe will develop the game levels and compile the final project documentation, ensuring clarity and comprehensiveness. Grace is tasked with designing the `Studio` class and main function, which encapsulates the game engine, providing the backbone for the gameplay loop. Wendy will manage `Block` behaviour, including their movements and the implementation of diverse `Block` types, ensuring robust and dynamic gameplay mechanics.

To support scalability and maintain modularity, we have chosen several design patterns. The **Observer Pattern** will synchronize graphical and text outputs, ensuring consistent updates between visual elements and internal logic. The **Decorator Pattern** will enable flexible extensions for `Block` functionalities, allowing us to introduce special behaviours efficiently. Lastly, the **Factory Method Pattern** will streamline `Level` creation, providing a clean and extensible way to handle different game configurations. Through this structured approach and collaborative effort, we aim to deliver not only a high-quality submission, but also a project that showcases innovation and ambition.

Questions

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?

To design our system to allow blocks to disappear after 10 additional blocks have fallen without being cleared, we can implement a mechanism that tracks all active blocks using a dynamic array or a similar data structure within the `Player` class. Each time a new block is generated, it is appended to the end of the blocks array. If the array's length exceeds 10, the oldest `Block` (at index 0) is removed to ensure only the 10 most recent blocks remain in the array. This removal is followed by shifting all subsequent blocks forward by one position to maintain the array's structure. This approach ensures that the array always reflects the most recent blocks while facilitating the removal of older, uncleared blocks. Additionally, we could optimize the design by using a circular buffer, which avoids the need for manual shifting and improves performance, particularly as the game scales. This modification would ensure efficient management of `Block` lifetimes, enhancing the gameplay experience by introducing a strategic element for players to clear blocks promptly.

How could you design your program to allow for multiple effects to be applied simultaneously?

To design the program to allow multiple effects to be applied simultaneously, we adopt an extensible object-oriented approach. Functions responsible for applying effects are

organized into relevant classes, such as `Studio` or `Player`, depending on their purpose. This modular design ensures that the implementation is clean and maintainable.

To enable simultaneous application of effects, we define separate methods for each effect. These methods can be invoked independently or in combination. For example, if two effects need to be applied concurrently, we call both respective functions within the same sequence of operations. This ensures flexibility in managing effects dynamically.

For scalability, if new effects need to be added in the future, we simply create a new function representing the effect and integrate it into the appropriate class. This minimizes code duplication and aligns with the principles of open-closed design, where classes are open for extension but closed for modification.

In our current implementation, the `force()` function resides in the `Player` class because its primary role is to manipulate the blocks that appear for a `Player`. Other effects, such as `render()` and `blind()`, along with utility functions like `clearLine()`, are encapsulated within the `Studio` class. This separation reflects the different responsibilities of these classes and promotes a clear division of concerns.

By structuring the code in this way, we ensure that the game logic remains extensible, allowing us to easily incorporate additional effects or adjust existing ones without disrupting the overall program structure.

How easily can you accommodate change?

Through a variety of methods implemented throughout our code, we can accommodate change quite easily. We would first start by eliminating magic numbers and replacing them with variables. It would make it easier for us to change different parameters, such as board scaling, probabilities, and rules, without having to change multiple lines of code spanning across various files. We will also design everything using classes, of which will have clearly defined purposes (i.e. a `Level` class, a `Block` class, a `Board` class, etc.). This allows for changes to be contained to the class affected, where if we were to add a new `Level` or `Block` type, other unrelated parts of the program would not be affected. Finally, we will focus on less coupling and more cohesion by reducing component dependencies. We will ensure each object created has a specific purpose, thus ensuring that any changes being made for a specific purpose can be paired back to its corresponding object instead of multiple.

Could the generation of such blocks be easily confined to more advanced levels?

Yes, the generation of blocks can be easily confined to more advanced levels using the **Factory Method Pattern**. This pattern allows the creation of objects (in this case, `Block`) to be delegated to subclasses, enabling a flexible way to change object creation logic depending on the game's level. For example, at lower levels, the factory can generate

simpler or predefined blocks, while at more advanced levels, it can generate more complex or varied blocks with different properties (like heavier blocks or different probabilities). By using a factory method, you can modify the `Block` generation logic without changing the rest of the game code, making it easier to scale and adjust the difficulty as needed. The creator in this case is the `Level`, and it would be changing the product, which is the possibility of each block appearing.

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

To design a program that accommodates the introduction of additional levels with minimal recompilation, we propose an architecture centred on abstraction and modularity. At the core of this design is an abstract base class, `Level`, which encapsulates the common interface and shared functionality of all levels. This abstraction acts as the foundation for extending the system. Each `Level` can be implemented as a derived class, overriding or extending the behaviour defined in the base class to reflect unique rules and mechanics. This approach ensures that new levels can be seamlessly integrated by simply defining a new class, implementing its specific logic, and registering it with the system—without altering the existing codebase.

To further minimize dependencies and recompilation, we plan to employ the **Factory Method Pattern** to handle the creation of `Level` objects dynamically. This pattern allows the program to instantiate levels based on configuration files or user input, reducing the need for hard coded logic. By externalizing `Level`-specific configurations, such as `Block` properties, scoring systems, or speed parameters, into easily editable files, new levels can be introduced or modified without requiring changes to the source code.

Additionally, the use of dependency injection ensures that components interacting with the `Level` interface remain decoupled from specific implementations. This decoupling enhances flexibility, as modifications to one `Level` implementation do not propagate across the system. To streamline integration further, we plan to design the game's `Level` loader to scan for new `Level` files at runtime. This eliminates the need for manual registration of new levels, promoting scalability and enabling developers to focus on creativity rather than infrastructure.

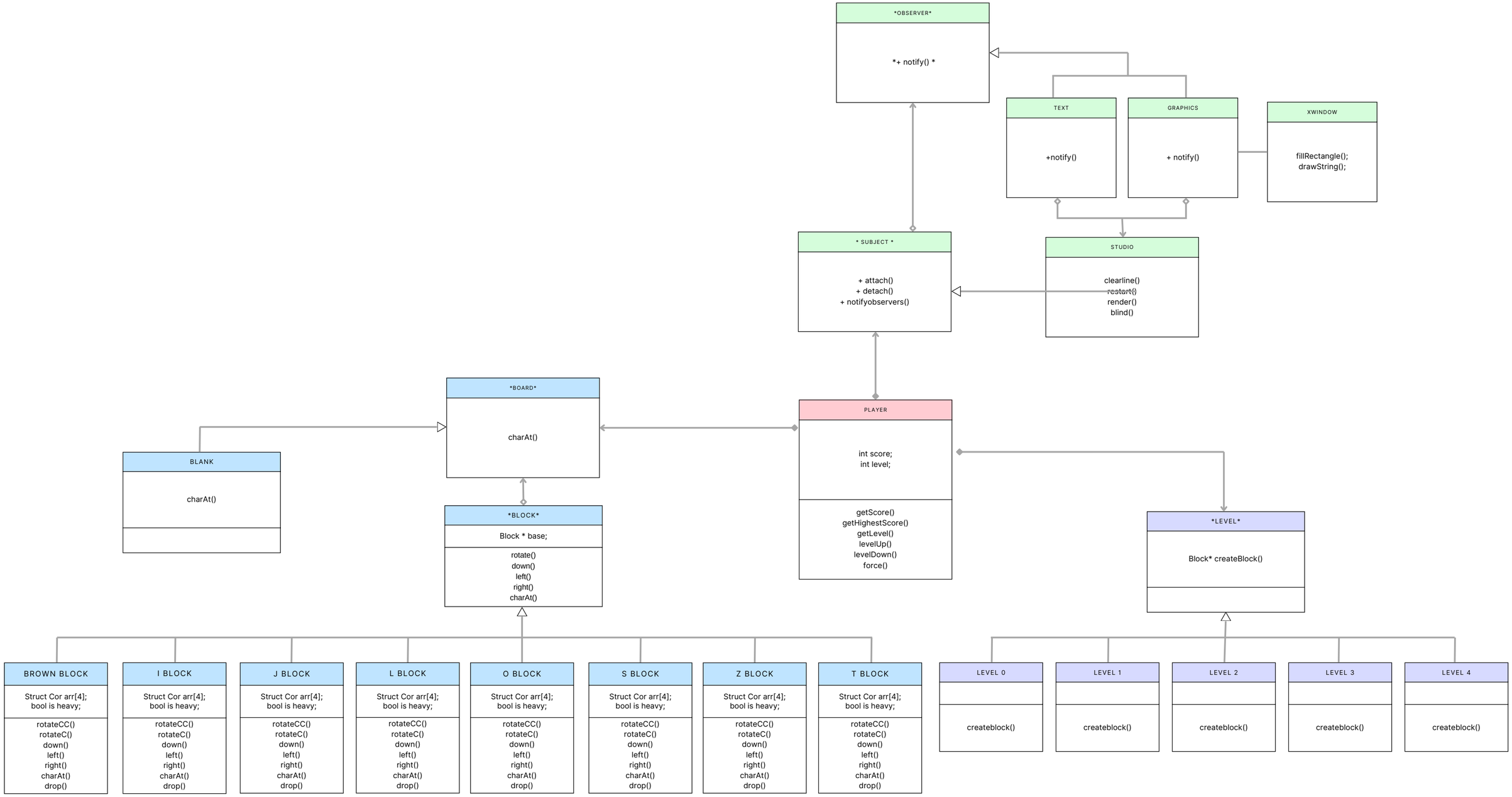
This modular and extensible approach not only supports the addition of new levels with minimal recompilation but also fosters long-term maintainability. It allows the system to evolve organically, empowering developers to continually innovate and expand the game with minimal disruption to existing functionality.

Can you prevent your program from having one else-branch for every possible combination?

Polymorphism in object-oriented programming allows objects of different classes to be treated uniformly via a common interface or base class. It lets you define a base contract while specific behaviours are implemented in subclasses, with the actual behaviour determined at runtime. For example, in Biquadris, you might have a base class `Block` with derived classes such as `IBlock`, `TBlock`, and `LBlock`, each implementing their own unique behaviour. Polymorphism simplifies code by eliminating the need for conditional checks (like if-else or switch statements) for each `Block` type, as the program dynamically determines the appropriate behaviour based on the object's actual class. This approach makes the code more extensible and easier to maintain, as adding new `Block` types only requires creating new subclasses without modifying existing code.

What if we invented more types of effects?

If more types of effects were invented, we would create a class called `Effect`, and create subclasses for specific effects (i.e. some subclasses would be `BlindEffect`, `FocusEffect`, etc.). As each effect acts independently, the program would maintain low coupling by decoupling the effects from the rest of the game. All effects would be managed in a centralized file system that is linked to the different blocks needed. For example, `BlindEffect` would be a class linked to a `Studio`, while `heavy()` could be linked to the `Block` decorator as a class which would oversee all blocks.


OBSERVER
*+ notify() *
TEXT
+notify()
GRAPHICS
+ notify()
XWINDOW
fillRectangle();
drawString();
* SUBJECT *
+ attach()
+ detach()
+ notifyobservers()
STUDIO
clearline()
restart()
render()
blind()
BOARD
charAt()
PLAYER
int score;
int level;
getScore()
getHighestScore()
getLevel()
levelUp()
levelDown()
force()
BLANK
charAt()
BLOCK
Block * base;
rotate()
down()
left()
right()
charAt()
LEVEL
Block* createBlock()
BROWN BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
I BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
J BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
L BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
O BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
S BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
Z BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
T BLOCK
Struct Cor arr[4];
bool is heavy;
rotateCC()
rotateC()
down()
left()
right()
charAt()
drop()
LEVEL 0
createblock()
LEVEL 1
createblock()
LEVEL 2
createblock()
LEVEL 3
createblock()
LEVEL 4
createblock()