Final Report

Overview

This report will explain the design and challenges faced working on the game Biquadris. The implementation of Biquadris involved several key aspects, each contributing to the overall functionality and user experience. The main function ('main.cpp') serves as the entry point, initializing the game environment and managing the game loop. Player turns are handled by the 'handlePlayerTurn' function, which interprets user input commands and updates the game state accordingly. Additionally, the 'Xwindow' class encapsulates the graphical interface using the X11 library, enabling the visual representation of the game board and blocks. Core game logic, including managing the game board and blocks, is implemented in the 'Board' and 'Block' classes, respectively. Different levels of gameplay complexity are achieved through subclasses of 'BlockFactory'. Notably, all memory management in the project was handled through smart pointers and vectors, ensuring efficient resource utilization and preventing memory leaks. This modular and well-structured design promotes code reuse, readability, and maintainability, facilitating the development and enhancement of the Biquadris game. Throughout the development process, various challenges were encountered, such as implementing smooth game mechanics, managing graphical rendering efficiently, and ensuring robust error handling. However, by adopting a systematic approach and leveraging the principles of object-oriented design, these challenges were effectively addressed, resulting in a functional and enjoyable implementation of Biquadris. The system was designed to accommodate potential future enhancements, such as additional levels of gameplay, with minimal recompilation by employing flexible class hierarchies and modular components.

Design

In our project's design, we began with an elaborate plan incorporating various patterns and structures to manage the game's complexity. Initially, our UML diagram included the Observer Pattern (Subject, Observer, and Virtual Block with dependencies) and the Decorator Pattern outlined (Block Special Action connected to Heavy and Force). Graphics Display had an

aggregation relationship with Windows, introducing unnecessary complexity. However, during implementation, we realized that some initial ideas were overly intricate. One significant change was removing the Observer Pattern and Decorator Pattern while retaining the Factory Method, initially included to manage communication between game components. We found it unnecessary and opted for a simplified approach relying solely on the Factory Method, reducing complexity while maintaining flexibility. Replacing these components with the Xwindow class simplified the structure and reduced coupling between modules, allowing for a more direct approach to graphics display. Leveraging the capabilities of the Xwindow class eliminated the need for the Observer Pattern and its associated classes, resulting in a more cohesive and modular design.

The cohesion and coupling of our program modules were crucial considerations throughout the project. We aimed for high cohesion, ensuring each module encapsulated a single, well-defined responsibility, such as the Block module handling individual game blocks and the Board module managing the game board's state and logic. Minimizing dependencies between modules promoted code reusability and maintainability. Regarding coupling, we aimed to minimize inter-module dependencies to enhance modularity and flexibility. Adhering to principles of abstraction and encapsulation, we reduced reliance on one module's internal details, facilitating easier modification and extension of individual modules without impacting the entire system. Furthermore, adjustments were made to the game board, introducing features like the 'blind' state and tracking the next block. These enhancements improved overall functionality and user experience. Integrating these features directly into the board class improved cohesion and made the code more organized and understandable. Additionally, some variables and sections in our UML diagram were renamed for clarity and consistency, ensuring uniformity throughout the project.

We simplified our project design from the initial Due Date 1 version by removing the Decorator Method, restructuring the Board class and streamlining special features like 'Blind,' 'Heavy,' and 'Force' actions. Previously tied to the Virtual Block, we refined these elements by integrating them into more suitable classes, enhancing clarity and reducing complexity. Specifically, we moved the 'Heavy' functionality to the Block class and relocated the 'Blind' feature to the Board class, aligning them with their respective responsibilities. Refactoring the

'Force' action allowed seamless integration into player interactions, resulting in a cleaner separation of concerns and a more streamlined solution that met project goals.

Resilience to Change

Ensuring adaptability to change is a core aspect of our project's design strategy. We achieve this through strategic use of the Factory Method design pattern, which allows seamless introduction of new game elements like blocks without disrupting existing code. For example, adding more levels to the game is facilitated by creating new subclasses within our BlockFactory class, preserving core functionality while incorporating fresh gameplay elements.

Furthermore, our codebase's modular structure plays a crucial role in its adaptability. Each game aspect, such as the game board, blocks, and special effects, is neatly encapsulated within its own module or class. This modular approach enables isolated changes or additions to specific game components without impacting the entire system. Introducing new block types with unique effects is simplified by creating new subclasses of the Block class, facilitating the expansion of game features.

Additionally, our design emphasizes abstraction and encapsulation principles, allowing for independent evolution of system components. The Board class, for instance, encapsulates the game board's state and logic, providing a clear interface for interaction while hiding internal details. This abstraction facilitates adjustments to the board's behavior or appearance without affecting other system elements.

Moreover, our design's flexibility enables seamless introduction or modification of features with minimal disruption. Whether it involves adding new special effects or tweaking existing ones, adjustments can be made to relevant classes responsible for these effects without causing widespread system effects. The loose coupling between components ensures localized changes, minimizing unintended consequences.

Overall, our project's design prioritizes flexibility, modularity, and abstraction, laying a robust foundation for resilience to change. Through the strategic use of design patterns,

modularization, abstraction, and encapsulation, we've established a framework capable of accommodating future enhancements or modifications to the program specification.

Answers to Questions

[Blocks] Question: How did you modify your existing design to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Were the generation of such blocks easily confined to more advanced levels?

New Answer: To implement the functionality where generated blocks disappeared from the screen if not cleared before 10 more blocks had fallen, we modified our existing design to incorporate this feature seamlessly into the game mechanics. Within the Board class or its relevant functions responsible for updating the board state, we implemented logic to decrement the expiration counter for each block on the board during each turn. Once the expiration counter reached zero, indicating that the block had exceeded its lifespan, the block was removed from the board. The design of the system included the following key aspects:

1) Expiration Counter Logic: Each block had an expiration counter attribute initialized upon generation. During each turn, this counter was decremented for all blocks on the board. When the counter reached zero, the block was removed from the board.

2) Handling Removal: Upon reaching the expiration counter threshold, the block was removed from the board. This removal was managed using a method like eraseBlock, if such a method existed in the code.

3) Adjusting Board State: After removing an expired block, we ensured that the remaining blocks were shifted down accordingly to fill the empty space left by the removed block. This ensured the integrity of the game board was maintained.

Yes, the generation of such blocks was easily confined to more advanced levels. By leveraging the Factory Design pattern present in the codebase, we customized block generations for different levels. This allowed us to configure higher levels to include special disappearing

blocks, aligning with the usual progression of games where new features are introduced as players advance.

[Next Block] Question: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

New Answer: In our program design, we implemented the Factory Method design pattern as planned to accommodate the introduction of additional levels into the system with minimal recompilation. By using polymorphism, we ensured the correct implementation of different level classes. This approach allowed us to seamlessly integrate new difficulty levels into the game without extensive code changes.

Initially, we proposed employing the Factory Method design pattern alongside a hierarchy of subclasses for each level to support the addition of more difficulty levels in the Biquadris game system. While the concept was similar, our initial design included a Levels class serving as the base, with subclasses representing levels 0 through 4. However, upon further implementation, we found it more efficient to directly implement level classes without a base class, simplifying the hierarchy.

Our final approach involved directly implementing individual level classes, each containing the specific logic for generating blocks tailored to its designated difficulty level. By leveraging polymorphism, the game dynamically selected the appropriate block generation logic based on the player's chosen difficulty level. This design allowed for seamless integration of new difficulty levels by creating additional level classes, without the need for extensive modifications or recompilation of the entire program. Additionally, we ensured that each level class could be easily customized to adjust block generation parameters, such as probabilities, randomness, or input file sequences. This flexibility in level design facilitated the creation of diverse gameplay experiences for players across different difficulty levels.

[Special Actions] Question: How did 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?

New Answer: In our initial design considerations, we planned to employ the Decorator design pattern to manage multiple effects simultaneously in the game. However, upon implementation, we found that the Decorator pattern introduced unnecessary complexity to our design. Instead, we opted for a different approach that allowed for the simultaneous application of effects without relying on the Decorator pattern. Our final solution involved directly applying multiple effects to the game state without the need for decorating objects with additional behavior. Each effect is handled independently, and conflicts between effects are resolved programmatically based on predetermined rules. Within the code, we structured the effects system to handle each effect as a separate module or class, with each responsible for its specific functionality. These classes encapsulate the logic for applying their respective effects to the game state. By decoupling the effects from the game objects themselves, we avoided the need for complex inheritance structures or decorator chains. For future expansions and the introduction of new effects, our design allows for the straightforward implementation of new effect-handling mechanisms without the need to create additional Decorator classes. By avoiding the Decorator pattern, we maintained a cleaner and more straightforward architecture while still achieving the desired functionality of managing multiple effects simultaneously.

[Command Interpreter] Question: 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. The board should be redrawn, both in text and graphically, each time a command is issued. For the graphic display, redraw as little of the screen (within reason) as is necessary to make the needed changes. For multiplied commands, do not redraw after each repetition; only redraw after the end. (Hypothetical Question)

New Answer:

Command Registry

Maintain a mapping of command names to their corresponding actions in a map like data structure. This mapping can be dynamically updated during runtime. Implement a command processing system that dynamically looks up commands from the map and executes their associated actions. This allows for additions, modification or removal of commands without recompiling the code.

Command Renaming

Extend the command mapping to support renaming. Allow users to specify the new names for existing commands, updating the map accordingly. This can be achieved by an additional rename command that takes two arguments: the old command name and the new command name.

Macro Language Support

Introduce macro language that allows users to define and execute a sequence of commands under a single name. Macros can be stored in the map along with regular commands, enabling them to be invoked like any other command.

Redrawing of the Board

Ensure the board is redrawn both in text and graphically after each command is executed. Use efficient rendering techniques to update only the necessary parts of the screen and minimize redrawing time. For sequences of commands, such as though with a repetition factor or macro language, redraw the board only after the entire sequence of commands have been executed.

Possible Alias complications

Command aliases introduce additional complexity as users are allowed to input as much of a command as needed to distinguish it from other commands. If we were to keep a map of commands to their prefixes needed to distinguish it from other commands, this would not be able to handle renaming commands during runtime. We would have to construct a DFA where each state represents a single user input character and define transitions to be the next state based on user input. The accepting states would be states where a valid and unambiguous command

prefix has been entered. To rename a command, we would have to modify the DFA during runtime by deleting transitions and states associated with the old command and adding transitions and states for the new command.

Extra Credit Features

In our project, we used smart pointers such as std::unique_ptr and std::shared_ptr alongside Standard Template Library (STL) containers like std::vector, std::map, and std::unordered_map to manage memory efficiently. By employing smart pointers, manual memory management complexities were mitigated, reducing the risk of memory leaks and dangling pointers. This shift in memory management responsibility to the compiler enhanced the overall robustness and reliability of the program. Additionally, STL containers facilitated the dynamic storage of game data structures such as the game board, block configurations, and player scores, ensuring efficient memory allocation and deallocation. Leveraging these containers streamlined memory management tasks, allowing the development team to focus more on implementing game logic and features rather than manual memory cleanup and resource management.