Biquadris Design Documentation

CS246 Spring 2023

Stephen Im, Daniel Lu, Soham Basu

Table of Contents

Overview of Classes

Class	Summary	HAS-A	OWNS-A
Board	The Board class is responsible for the movement of the blocks on the board and retrieving information about each tile.	**N/A**	**Block** **Level**
Level	The Level class is responsible for deciding the type of a new block, reading from files if necessary.	**N/A**	**N/A**
Block	The Block class is responsible for defining the various block types, and constructing them.	**N/A**	**N/A**
Subject	Manages the collection of observers	**Observer**	**N/A**
Observer	Responsible for notifying its subclasses	**N/A**	**N/A**
Graphics	The Graphics class is responsible for displaying all aspects of the game graphically.	**N/A**	**N/A**
TextDisplay	The TextDisplay class is responsible for printing out the current state of the game, including the boards, levels, and scores.	**N/A**	**N/A**
Xwindow	The Xwindow class is responsible for the backend of the graphics, initializing all colours etc..	**N/A**	**Graphics**
Trie	The Trie class is responsible for returning all possible autocompletions of any string command.	**N/A**	**N/A**
Cell	The cell class is responsible for keeping track of which blocks are currently on the board	**N/A**	**N/A**
Decorator	The decorator class is responsible for maintaining the decorating subclasses	**absBoard**	**N/A**
absBoard	Abstract base class used to organize the Decorator pattern.	**N/A**	**N/A**
Blind	Blind class is responsible for creating the blind special effect.	**N/A**	**N/A**

IBlock
+char getType()
+string firstRow()
+string secondRow()
JBlock
+char getType()
+string firstRow()
+string secondRow()
LBlock
+char getType()
+string firstRow()
+string secondRow()
OBlock
+char getType()
+string firstRow()
+string secondRow()
SBlock
+char getType()
+string firstRow()
+string secondRow()
ZBlock
+char getType()
+string firstRow()
+string secondRow()
TBlock
+char getType()
+string firstRow()
+string secondRow()
starBlock
+char getType()
+string firstRow()
+string secondRow()
Block
-vector<vector<int>> pos
-vector<int> lowerLeft
-int cellsLeft
-level
+char getType()
+string firstRow()
+string secondRow()
+int numCells()
+void clearCell()
+int getLevel()
+vector<vector<int>> getPos()
+unique_ptr<Block> blockGen(char, int)
Cell
-Block *block
-char type
+void updateBlock(Block *, char)
+char getType() const
Block *getBlock() const
Subject
- vector<Observer> *observers
+ attach(Observer *o)
+ detatch (Observer *o)
+ getState()
+ getState(int row, int col)
#notifyObservers ()
Observer
+notify()
Level0
-vector<string> names
-vector<ifstream> files
-int level
+unique_ptr<Block> newBlock(int currPlayer)
Level1
-int level
+unique_ptr<Block> newBlock(int currPlayer)
Level2
-int level
+unique_ptr<Block> newBlock(int currPlayer)
Level3
-int level
-ifstream file
+unique_ptr<Block> newBlock(int currPlayer)
+void getFile(string)
+viud setRandom(bool)
Level4
-int level
-int count
-string name
-ifstream file
+unique_ptr<Block> newBlock(int currPlayer)
+void getFile(string)
+void setRandom(bool)
Level
-absBoard *board
-bool random
-int level
-unsigned int seed
+unique_ptr<Block> newBlock(int currPlayer)
+void getFile(string)
+setRandom(bool)
Board
-int numRows
-int numCols
-vector<vector<Cell>> theBoard
-absBoard *board
-bool blindActivated
-bool blockDroppedAfterBlind
-bool heavy
+char getState(int row, int col)
+char tileAt(int row, int col)
+void render()
+void rotateBlock(Block *block, bool clockwise)
+int shift(int x, int y, Block *block, bool drop)
+bool swapBlock(Block *oldBlock, Block *newBlock)
+void putBlock(int col, char type)
+int clearRows(vector<int> &playerScores, int currPlayer)
+bool isHeavy()
+void setHeavy()
+void triggerBlind()
+bool isBlind()
Graphics
-unique_ptr<Xwindow> window
-Board *b1
-Board *b2
-vector<int> levels
-int currPlayer
-vector<int> scores
-vector<Block*> &queue1
-vector<Block*> &queue2
-bool blind
+notify()
+void updateBoard(vector<int> levels, vector<int> scores, bool blind)
TextDisplay
-Board *b1
-Board *b2
-void printHeader(string, int)
+vector<Block*> queue
+vector<int> currLevel
+int currPlayer
+void display(const vector<Block*> &queue1, const vector<Block*> &queue2, vector<int> currLevel, vector<int> currScore, bool p1blind, bool p2blind)
+notify()
TrieNode
- int size
- int words
- bool isWord
-unique_ptr<unique_ptr<TriNode>[]> letters
-void updateWords()
+void insert(const string &word)
+void remove(const string &word)
+void updateSize()
+vector<string>find(const string &word) const
+int getSize()
Decorator
- absBoard *component
+char tileAt(int row, int tile)
absBoard
- const int COLUMNS
- const int ROWS
+char getState(int row, int col)
+char tileAt(int row, int col)
+void render()
+void rotateBlock(Block *block, bool clockwise)
+int shift(int x, int y, Block *block, bool drop)
+bool swapBlock(Block *oldBlock, Block *newBlock)
+void putBlock(int col, char type)
+int clearRows(vector<int> &playerScores, int currPlayer)
+bool isHeavy()
+void setHeavy()
+void triggerBlind()
+bool isBlind()
Xwindow
-Display *d
-Window *w
-int s
-GC gc
-unsigned long colours[10]
+ void fillRectangle (int x, int y, int width, int height, int colour)
+ void drawString (int x, int y, string msg)
Blind
-bool blockDropped
-char character
+tileAt(int row, int col)
+void setBlockDroppedTrue

Updated UML

Changes since DD1:

- We realized that the two special effects Force and Heavy should not really be incorporated as a part of the Decorator design pattern, so we have omitted the classes entirely.
- We realized that to implement the auto-completion feature in a clean and organized fashion without any hardcoding, the Trie class was needed so we added that.
- We had some troubles maintaining privacy of our Block and Board class without running into issues involving the redoing of our entire design, so we thought it’d be easiest to just incorporate a new class Cell, which kept track of whether a block is completely wiped off the board or not.

Design Patterns / Resilience to Change

Observer Design Pattern (TextDisplay, Graphic)

- Our TextDisplay and Graphic classes make use of the Observer pattern, both with a notify() method, which notify the observers of any changes that are made, and reflect the changes in the display accordingly.
  The Graphic class makes use of the Xwindow class that had been introduced in A3, with changes to match the desired design that we were looking for.

Decorator Pattern:

- Our Decorator, absBoard, Board, and Blind classes make use of the Decorator pattern to show the blind special effect on the player's board, which partially blocks their view of their board.

Factory Method Pattern:

- Our level and Block class make use of the Factory method pattern. Each level is equipped with a method getNextBlock, which determines what block should be spawned next for that specific player on the current level. And so, adding new blocks is just a matter of adding that method into the new block class.

Polymorphism and Inheritance:

- Our Board and Decorator classes inherit from our absBoard class, and our Blind class inherits from our Decorator class. This design is in place to take advantage of polymorphism when it comes to displaying special effects such as the blind effect on a player's board.
- The Level0, Level1, Level2, Level3, and Level4 classes all inherit from our Level class. This design allows us to use common methods such as newBlock to generate a new block type, which is implemented differently depending on the specific level class it belongs to. Hence, we are able to make necessary adjustments to specific Level classes, while keeping some common methods the same across all 5 subclasses.
- The IBlock, JBlock, LBlock, OBlock, SBlock, ZBlock, TBlock, and starBlock classes all inherit from our Block class. This allows us to have common fields for each block, while allowing us to adjust the constructors to reflect the different shape of each block.
- The GraphicDisplay and Text classes inherit from our Observer class. This allows us to use notifyObserver methods within our Observer class to update our GraphicDisplay and Text classes, each of which has their own way of displaying the current state of the game.

Answers to Questions

Question: 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?

Answer: The block generation process could be altered to create a new special block type that disappears after a certain number of turns. Each block could also contain an “age” field that indicates how many turns ago it was generated, with the special block’s shape being set to empty once a certain number of turns have been reached.

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

Answer: Use doubly linked list, if new levels are introduced, update the [back] pointer in the list object, and we don’t need to worry about what the current level is, just go to the Level node and let polymorphism take care of it.

Question: 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?

Answer: Through the use of the decorator pattern, we are able to accommodate for the application of multiple effects simultaneously. If more effects were invented, we would simply need to create separate subclasses for each of them and have them inherit from the Decorator class. The decorator pattern also eliminates the need of having one else-branch with every possible permutation of the different effects.

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.

Answer:
The system could be designed with a universal testing harness that reads commands, and then calls the appropriate methods and functions. Commands could be renamed by updating their names once in the universal testing harness. Furthermore, to allow users to rename existing commands, the command names could be stored as variables in the universal testing harness, rather than string literals. This would allow us to create a new command called “rename” which could alter these variables, hence changing the names of existing commands.

Extra Credit Features

- The sole “extra feature” that we implemented was the use of smart pointers–specifically only unique pointers as we wanted to be extremely careful and prioritized efficiency. We did not use any memory allocation, and deletion throughout our entire code. It was difficult to keep track of what classes and objects owned each pointer, and ran into many segmentation faults due to smart pointers going out of scope, without us realizing. With a lot of trial and error and code tracing, we were able to pinpoint where these errors were occurring to get around them.

Final Questions

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

- This project taught us that communication is crucial to developing software in teams. Communication, task splitting, and using a universal codebase where we can pull, push, and merge code such as GitHub.
- We learned that it’s also extremely important to comment and keep our code properly formatted, because other people will not know what self-written code will mean and taking time to explain to others for even the simplest functionalities takes up a significant amount of time

What would you have done differently if you had the chance to start over?

- If we had the chance to start over, we would have organized our time better so that we efficiently move through each portion of the project, leaving us with a fair amount of time at the end to fully test, and add additional features to our project.

Conclusion

In conclusion, we all had a wonderful experience partaking in this project. It was a thrilling experience seeing everything we learned in CS246 throughout this term come together. We all gained valuable experience in object-oriented software development, working in a team, and using various software development tools that we had previously learned in CS136L. It was very fulfilling seeing the project come into fruition after the countless hours and late nights we spent debugging the many, many segmentation faults we experienced.