Plan of Attack

By Alexandru Camer, Karem Abdul-Samad, and Zhao Gui

Table of Content:

Specification Questions

***Question**: After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a gameboard? Why or why not?*

Answer: The Observer Pattern would indeed be a good pattern to use when implementing the gameboard. Redrawing the gameboard at every step would be computationally inefficient as it would require redrawing all 40 tiles and all players at each step in the game. As a result, by making the gameboard an observer on the players and tiles (subjects), when a player is moved or a building is improved or any other visual updates..., these subjects can just notify the gameboard that a change occurred and the gameboard would simply redraw itself (or perhaps only redraw the changed parts). This offers a much more efficient solution to keeping the display up to date as we only update the display when a display-related change occurs. Therefore, the observer pattern is a good fit for the implementation of a gameboard.

***Question**: Is the Decorator Pattern a good pattern to use when implementing improvements? Why or why not?*

Answer: Decorator Pattern is not a good pattern to use when implementing improvements. The Decorator Pattern is intended to let you add functionality or features to an object at run-time rather than to the class as a whole, however, in this case, improvement costs don't necessarily scale as a well-defined function of the number of improvements, so utilizing the decorator pattern would require a lot of hard coding to determine relationships between the current improvement and the next. Furthermore, since we know there are at most five improvements in total, it would be easier to just provide improvement details at object construction and not worry about it later. In conclusion, it would not make a lot of sense to use a decorator pattern.

***Question**: Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?*

Answer: Template method will come in handy when trying to implement needles halls and SLC more closely to Chance and Community Chest cards. A template method defines the steps of an algorithm in an operation, but lets the subclasses redefine certain steps, though not the overall algorithm's structure. At the Template parent class, we would have a template function with potential operations that the chance and community cards could have on a player (move player or withdraw/deposit cash into player). These potential operations would be virtual and the subclasses for each card would implement these potential operations with respect to the card's effect. Another benefit to the template method is that we can customize the string the player reads upon card pickup without changing how the card works or is implemented.

Project Breakdown

Design Explanation:
Our “master” class is the game class. We’ll tie all our players and board (of tiles) to this class for simple memory management (according to RAII principles).

We decided to follow a display model similar to A4Q3 for the game of life. Our goal was to find a design pattern which would update the gameboard display anytime a display-related change occurred in the game model. The obvious choice for a design pattern in this case was to use the observer pattern, where the display (observer) observes the game class, and some of its members, specifically players and tiles (subjects). This way, whenever a subject changes (e.g. whenever a player moves, or whenever a tile receives an improvement or…), the changed subject would notify the board, which would then update itself. This use of observer pattern increases our program efficiency since we only update our display when a display-related change occurs, but comes at the cost of increased complexity in our code.

A considered alternative method was to use the polling technique. The way this would work is that, every time a player inputs the ‘next’ command, the display will iterate through the entire board to update its info on each tile. This is because, in this case, the display doesn’t know if a change occurred at any given step, so it needs to update its information on every tile just in case a change occurred. This method is typically avoided in the cases where there are many objects to poll, as it would have a significant cost on the efficiency of the system. However, with only 40 tiles and several players needing to be updated by the display, it is not a bad idea for this specific application. The main benefit of using this is that it would reduce complexity in our code. Though, overall, we still believe the increased efficiency of the observer pattern outweighs the simplicity of the polling technique.

For the tiles we wanted to follow good design practices by abstracting the tiles in a way that no code inside the player would be tile-specific. However, we noticed that all the tiles fell into 2 categories: Those you can own, and those you can’t own. As a result, we decided to make an abstract tile class, and 2 abstract subclasses; property, and non-property. Property consists of tiles a player could own, and the non-property cannot be purchased by the players. This way, if the player lands on a non-property tile, it simply calls some void “visit” method which will apply

the tile's effect on the player. On the other hand, if the player lands on a property tile, things like purchasing the property, auctions and tuition costs will be considered as needed automatically.

For our tile container, we will be implementing a board class, which is essentially a wrapper class for a vector of pointers to tiles (Note we tie the tiles to the board class for simple memory management according to RAII principles). This board class will be accompanied by a board iterator. Though, this board iterator will be slightly different than that of a typical iterator design pattern. This board iterator will have a cyclic feature, meaning that once the iterator reaches the end of the vector of tiles, it will automatically be set to point to the beginning of the vector, i.e. the iterator will never end. This feature makes the iterator an ideal way for the players to move around the tiles of the board. So, by giving the player class a board iterator, whenever a player is moved, we would simply just increment the player's board iterator by the desired amount and then call the "apply effect" function of the landed-on tile on the player itself. This is also good practice since the player technically doesn't know anything about the board class' implementation, aside from the fact that it represents a circular sequence of tiles.

For common utility methods we will be making a utility library. This will include methods such as rolling dice, possibly an auctioneer class, and other common methods. It will also include some useful constants, such as the terminal codes for specific colors on the board, some common string phrases to be used by the controller, or setting a max number of players etcetera.

Currently, we plan to let the client handle command interpretation. So, the client will mostly call methods of the game model, which in turn will handle the correct logic details. The I/O class will specifically handle saving and loading into and out of files respectively. We made this decision because we realized that we will need to make our own spec for how a save file looks exactly, and we wanted to move these methods into their own "I/O"-related class. However discussions have revolved around if this is necessary. Ideally, we want only one place where any input and output (including to std::cout) is done, so we might change this if we find a cleaner method which easily allows for this.

Meetings:
We plan to meet up on Thursday August the 6th, and do the work together. We have separated the works as outlined at the bottom of the document. We aim to finish most of the layout of the code done on the 6th (header files and such), so that we can individually work on the tasks on our own. If we realize that we need further help as a group effort (perhaps for outliers like the fileIO class) we will meet up again to finalize the work. We will be taking advantage of git branches to easily remove code which causes the program to crash or not compile. This way we always have a functioning version in the main branch. Lastly, we will write the final report on August 14th. Please see below for the detailed Plan and work assignment.

Milestones:

Mile-stone	Description	Expected Comple-tion Date
1	Having a functioning board where players can be added and move across the board. Tiles are not expected to produce any behavior yet but players should be able to roll the dice and move.	Aug 6
2	Implement the tuition mechanic where players can buy and own tiles, and receive payments when other players land on those tiles they own.	Aug 6
3	Implement the monopoly section for the academic tiles. The program should be able to know what tiles belong to what group, and if a monopoly exists. The improvements do not need to be yet implemented.	Aug 6
4	By now the program should be able to recognize groupings of tiles, and should allow players to build improvements on those tiles.	Aug 6
5	Implement the residences and gyms, with all their spec done accordingly	Aug 7
6	Implement the non ownable properties regarding any cash moved to or away from a player.	Aug 7
7	Implement the tims cup mechanic, including finally forcing players in the DC tims line when appropriate.	Aug 7
8	Implement both SLC and Needles Hall tiles.	Aug 9
9	Implement the mortgage mechanic.	Aug 9
10	Implement the auction mechanic.	Aug 9
11	Implement the trading mechanic between players.	Aug 9
12	Implement the bankrupt mechanic.	Aug 9
13	implement the save and load features of the specification.	Aug 9
14	Add the testing mode to the program.	Aug 11
15	Finish the final classes, and the MakeFile, and write the final report.	Aug 11
16	Debug program and add exception safety mechanisms to prevent program from quitting abruptly.	Aug 13

17	Use the GTK library to replaceTextDisplay class with a GraphicDisplay	Aug 15

Note: This milestone layout was written in a way that targeted the most important features first, however some milestones can be completed asynchronously from each other. As a result we do not expect every milestone to be completed chronologically. Rather, these milestones will be our general guideline.

Work Division:

Name:	Classes:
Alex	Game, View, Observer, Subject
Karim	Player, Board, BoardIterator
John	Tile and all its descendants classes.
All	Game I/O