Plan of Attack: BuildingBuyer7000

By: Galen Wray and Nisarg Patel

Day-to-Day Plan

Date	Task	Person	Notes
Tuesday, November 18th	UML	Galen & Nisarg	
Wednesday, November 19th	Plan of Attack	Galen & Nisarg	
Friday, November 21th	Square and Subclasses	Galen & Nisarg	Set up abstract base class and corresponding subclasses, which represent the various properties and other squares on the board.
Saturday, November 22st	Player and Human	Galen & Nisarg	Create abstract player class and corresponding human subclass for gameplay.
Monday, November 24st	Computer and AI	Galen & Nisarg	Finish the final subclass of Player by implementing artificial intelligence.
Tuesday, November 25th	Gameboard and Text Display	Galen & Nisarg	Create Gameboard and Text Display classes to allow for movement around the board.
Wednesday, November 26th	Main Function	Galen & Nisarg	Read in saved gameplay and facilitate user-program interaction.
Friday, November 28th	Final Revisions and Debugging	Galen & Nisarg	Attempt to play the game and resolve any problems encountered.
Saturday, November 29th	DLC and Final Design Documentation	Galen & Nisarg	Finalize the design document, update UML and attempt to implement additional features. (eg. graphical display, adding house rules, etc.)

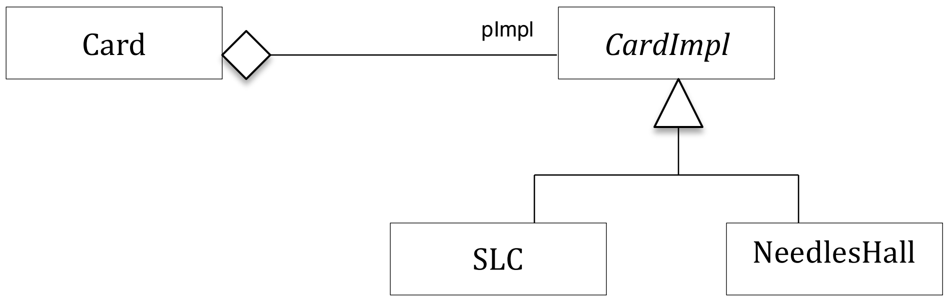
Questions

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

The Observer Pattern would be very useful when implementing the visual representation of the gameboard. Although many components of the display remain consistent throughout the game, two elements, improvements for academic buildings and representations for the location of each player, must be updated and changed. Rather than updating the entire display by checking the locations of each player and the improvements of each academic buidling before reprinting the board, it would be more efficient if the display was notified so that only the changed components were modified. For that reasion, using the Observer Pattern with the academic buildings and players as subjects and the display as the observer will allow for a more efficient update of the visual display.

2. 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?

If we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards, SLC and Needles Hall would both be different implementations of the same object: a card. As a result, the Bridge Design Pattern would be a good pattern to use. We would have a structure with the interface details of the card, a class for implementation with a pointer to the interface, and two subclasses, the SLC and Needles Hall classes, that inherit from the implementation class. Then, both SLC and Needles Hall would implement the card in their separate ways.



3. What could you do to ensure there are never more than 4 Roll Up the Rims cups?

In order to ensure a maximum of 4 Roll Up the Rim cups, a static integer counter can be used in the Roll Up the Rim cup class to count the number of cups as they're being created. Before creating a new cup, the constructor will verify whether or not the maximum number of cups are in already in play, and only if there are less than four cups will a new one object be created. Furthermore, when a cup is deleted the counter will be decremented to maintain an accurate number of active cups.

4. Research the Strategy Design Pattern. Consider the Strategy and Bridge design patterns, would either be useful in implementing computer players with different levels of difficulty/intelligence?

The Strategy Design Pattern would be an effective pattern in the implementation of a computer player with different levels of difficulty/intelligence since its main purpose is to abstract the behaviour of a class. In this case, we would be abstracting the parts of the behaviour of the computer player in separate classes, each corresponding to the different levels. At run-time, we would be able to select which of the different classes to execute based on which level was selected. This pattern would be more useful than the Bridge Pattern since the Bridge Pattern is used more for structural abstraction with the purpose of having the implementation and interface separately.

5. Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

In this circumstance, the Decorator Pattern would not be beneficial to implement. Although the improvements add functionality during runtime, which is the reason we implement the Decorator Pattern, they are only minor changes that are more simple to implement without it. In order to implement improvements and change the necessary functionality, it would suffice to use counter for the number of improvements for each academic building. With a counter, we can easily change the tuition amount for a building by using an array for tuition, and the number of improvements as the index. We will also be able to change the number of improvements in the visual implementation with only a counter, and adding an additional 'I' for each improvement. For these reasons, the Decorator Pattern is not needed for an effective implementation of improvements.