**Plan of Attack**

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 we encounter. |
| 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.) |

**Question:** 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 a good pattern to use when implementing a gameboard, particularily for the maintanence of the text display. Both the academic buildings and the players need to be updated on the display when either the improvements of buildings need to be added or removed, or the players need the be moved on each turn. As a result, the buildings and players serve as good candidates for subjects, notifying the text display, the observer, whenever their states have changed. The Observer Pattern is a good way to keep track of what needs to be changed on the display, rather than updating the entire display of before reprinting.

**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?

* Idk

**Question:** 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 check to 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 cup object be created. Furthermore, when a cup is deleted the counter will be decremented to maintain an accurent number of active cups.

**Question:** 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?

* strategy design pattern - abstracting the behaviour of a class - determined at runtime?
* ie. would have a different class for behaviour for the different levels
* http://stackoverflow.com/questions/464524/what-is-the-difference-between-the-bridge-pattern-and-the-strategy-pattern
* http://game-engineering.blogspot.com.tr/2008/07/bridge-pattern-vs-strategy-pattern.html
* bridge pattern is for structural abstraction (eg. hiding away implementation details and having no compilation dependency in the main class)

**Question:** 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 benefitial to implement. Although the improvements add functionality during runtime, they are minor changes that are simple to implement without it, where

First of all, adding improvements changes the text display, by adding ‘I’ to a property to represent each improvement. However, each improvement can be represented in the same way, so a counter for the number of implementations would suffice. Finally, the number of improvements changes the tuition of the property. This feature is also easy to implement without the use of the design pattern, since

* No
* While the improvements add functionality at run-time, all improvements are minor, and have little need for change (only a small change the the text display (I) and of the tuition) -> much simpler to keep track of the improvements all together (a counter for the number of improvements applied)
* Had we had different improvements for each level of improvement, the decorator pattern would have been advantageous