Plan of Attack:

Component	Deadline	Person Responsible
Black-box Test Cases	7 - 8 August	ALL
Loading and Saving games, I/O	10 August	Catherine Wang
Command Interpreter (main)	10 August	William Wang
Unownable	10 August	William Wang
Buying Properties	10 August	Trung Nguyen
Board	10 August	Catherine Wang
Square	10 August	William Wang
Makefile	10 August	William Wang
Player	12 August	Trung Nguyen
Ownable	12 August	Trung Nguyen
Transaction	12 August	Trung Nguyen
Auction	12 August	Trung Nguyen
Demo	13 August	ALL
Final Design Document	14 August	ALL

Our plan of attack focuses on finishing the “when all else fails” components earlier so that we will definitely submit something that runs upon DD2, in addition to the Makefile required for compilation. The prioritized “when all else fails” components consist of loading and saving games, the command interpreter, unownable squares, etc. After finishing them, we’ll focus on more complicated components such as trading and auctions and finally bring everything together after August 12th. We will take three days to work together to integrate all components and finish our design document and demo files. We will also write a few general test cases before starting to code (i.e. our black-box tests) to test the complete program, these tests will ideally be written by August 8th.

Our tentative UML diagram consists of the GameState class, used for saving the game. The Board class is responsible for creating and storing the 40 squares and up to 8 players in the game. The Square class itself is an abstract base class, which has the specific subclasses Ownable and Unownable. Ownable consists of the properties that can be owned, specifically Academic buildings, Residences, and Gyms. Unownable consists of the properties that cannot be owned, specifically the TimsLine, the SLC, and squares related to monetary services (called MonetaryServices).

In addition, we created Transaction and Auction classes used by Players and MonetaryServices; these classes regulate auctioning, trading, buying / selling properties and improvements, mortgaging, and paying the Bank or paying Players.

Designing the UML diagram was quite difficult. Determining where and how to store certain data (e.g. in a vector or map) was a source of confusion, but also considerable discussion. Specifically, a problem we faced was deciding which class should modify the state of others. For example, there was significant debate over how owners should be tracked during transactions between properties. While we have decided upon a tentative solution, this is subject to change if we encounter difficulties while implementing so we prepared a backup solution if needed. Another problem we encountered was how to implement saving / loading games and properly executing command-line arguments. We are not certain of how we will implement saving a game to a text file (one idea was possibly writing an input and output operator for GameState), so this is something we will consider and discuss before starting to code.

The TimsLine square was an interesting square, as we discussed ways to keep track of the players on TimsLine. Again, the issue of whether a vector or map should be used came into play. We needed to keep track of the number of rolls each player (if they are in capture list) made while on TimsLine, and create a Transaction object if someone needs to bail themselves out. We decided to store this information in a captured map, which contains keys of char (associated with players) mapped to the number of rolls they made so far. Each time a player begins a turn on the TimsLine square, the Board object checks if the player is in the captured map and proceeds accordingly.

In order to generate events with appropriate probabilities for the Needles Hall and SLC buildings, we simply had to choose a “magic number” to serve as the number for which

a TimsCup would be granted (provided that enough are left), from 1 to 100. Should the TimsCup roll fail, a second number would be generated, with probabilities matched appropriately. This seems relatively simple, though deciding what to do based on the numbers drawn may require more work.

The updateByTrans methods in the Board are for the purpose of updating the state of the game based on transactions, specifically updating owners of buildings. There has been significant debate on the subject as to whether the methods would be useful. These methods may be removed as the program is implemented.

Another challenge we anticipate during implementation is putting together components of a large-scale project, as none of us have worked on a project of this size. A significant issue is actually ensuring that all team members can access the final product as a single unit. To best prepare for this, we will study Git tutorials independently and leave several days to debug and integrate separate components. Furthermore, integrating separate components requires clear communication between group members. Each member should be familiar with the design of others' classes so that one can call methods appropriately while writing their own classes and functions. We plan to notify each other when we finish components and whenever someone is faced with issues, especially given the individual events going on during the final exam period.

Questions:

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

No, because board Square objects do not frequently change state and thus do not need to constantly notify other objects on the gameboard, so the Observer Pattern is not necessary. As well, players do not need to be notified when the state of a building changes because of its interaction with another player, since this does not directly affect their own gameplay. For example, a transaction only needs to be known by parties involved in the transaction. Any consequences of the transaction (e.g. increased tuition due to improvements) are only relevant to the player once they land on that square, so they do not need to be immediately notified of the transaction.

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?

Of the studied design patterns, the Iterator Pattern is most suitable for implementing Chance and Community cards. Since a stack of cards is best modelled by a stack data structure such as a linked list, an iterator can be used to traverse the stack of cards while they are drawn. Once the iterator reaches the bottom of the stack (i.e. the last element in the list), the cards are shuffled (with a separate algorithm) and the iterator is reset to the new first card in the stack.

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

The Decorator pattern is optimal for adding features to objects that require a series of different decorations (e.g. a text processor), represented by a linked list of decoration objects. In this case, the improvements do not add functionality to buildings, but simply increase tuition cost, so the Decorator pattern is likely not appropriate. As well, there's only one way to 'decorate' a building by increasing tuition, so having concrete decorator classes with different operations isn't necessary. Data associated with improvements (i.e. number of improvements, tuition with improvements) can simply be stored in the building objects.