Plan of Attack: DD1 Submission

Manit Bhatia, Vansh Jain, Rhythm Chawla

Component Breakdown and Deadlines

Component	Deadline	Person
UML Diagrams	March 18	Manit, Vansh
Project Plan	March 18	Vansh
Non Building Logic	March 19	Manit
Controller Logic	March 19	Rhythm
Cell and Building Component	March 19	Vansh
UML Updates more classes	March 20	Team
Internal Review & DD1 Finalization	March 21	Team
Display Components	March 22	Manit, Rhythm
Transaction Component	March 22	Vansh
General Debugging	March 22	Rhythm

Documentation Drafting for DD2	March 24	Team
Additional Component Development	March 24	Vansh
Load Game and Auction	March 24	Vansh
GameBoard Component	March 25	Manit
Initial Compilation & Testing	March 26	Vansh
Testing & Debugging	March 27	Team
Documentation Finalization	March 27	Rhythm
Final Code Refinement	March 28	Team
Bonus Features (if time allows)	March 29	Vansh, Manit
Final Testing and Playing	March 30	Team
Final Review & DD2 Submission	March 31	Team

Plan Execution

Our plan of attack is to have the basic framework of the game up before DD1. If we have all the basic functionality implemented, we can work up our way towards more complex functions, which will ensure safe expansion on our project. We want to make sure DD2 has a great working game, and this approach gives us time to tackle the unexpected errors that occur during testing as well. We also plan on taking on the bonus, and have brainstormed the idea of creating a bot, a computer that players can play against.

We have attached a tentative timeline above, and plan to adhere to it till the end for the smooth execution of our project. This also gives us time to debug, and be prepared for creating a project of such size and complexity.

UML Design Approach

We follow a Modular approach in UML, all set to adapt smoothly with the requirements in the near future. Our UML followed a bottom up approach, making it easier to plan inheritance of classes, abstract classes and respective subclasses. This UML will definitely change over time, and making the first version helps create a road map for us, so we can start coding our project with a mind map in place.

We modeled our UML based on few components that drive the game functionality. Class loadGame will handle game state details (saving and loading). In addition, the instance of GameBoard class will create and keep the game elements: cells and player classes. An abstract base class Cell will be defined, alongside subclasses Building and NonBuilding controlling the handling of each type of cell. Classes like Transaction and Auction handle economy and rights exchanges within the game.

We are planning to create a skeleton code for all the classes. This will give us an idea as to how many classes and subclasses will be there. Once we are done creating the skeleton, we will start expanding functions from bottom to top. Following this unconventional approach shows us the importance of each and every function above. This helps us set the fields well, and maximize cohesion since we will know which functions require which modules. Moreover, this approach also leaves room for decorators, and additions that might be required to add to our codebase.

Challenges and Considerations

Designing the UML model posed several challenges, especially in deciding what's the right classes to choose, what flow our code should have, which block connects to which etc. Once we were able to map out an empty UML structure, we decided to fill it in with functions and code them out. This will give us time to fill in the right functions, and if we need to change anything we can do that.

We are anticipating challenges of many sorts in this project. Firstly, none of us have worked on an decorator and observer patterns before and a project which is this complex. We also need to polish our skills on Git, and manage our repository effectively. We want to avoid conflicts to the best of our abilities, and working on this project gives us an opportunity to practice real world skills like these.

Another challenge is deciding how many modules we will need, and how to minimize the usage of extra functions. We want to be smart about our code, and make sure that lesser code can have more impact on the game, which will make the game run faster and the module pack smaller as well.

We are planning to make a bot for bonus. Figuring this out is going to be very interesting and will certainly pose a challenge to us. We also want the graphics to look as modern and sleek as possible, and we will try to make the board as informative and compact as possible, without ruining the look of our project.

Final Considerations and Testing Approach

To ensure reliability, we will test the project intensively. Our method of testing is going to be slightly interesting. Once we are done with the game, we will try playing it together, and figure out bugs from there. We will also hand a running interface to some of our friends and ask them to try and shut our program down using the command line. Making a program robust and prepared for all sorts of situations is one of the biggest goals of our project. We want to make a beautiful game. To make the game run fast, we will try making functions which are efficient to the best of our abilities.

Other final considerations would include testing each and every part of the code. We will then match our UML to the code, and try seeing if anything can be done better, to get the same functionality out of it. Once we are ready with the project, we will brainstorm more bonus ideas, because we already love the idea of creating Watopoly, and adding more features to the game will personalize our coding experience!

In conclusion, we are very excited about this project, and will try to test and implement its functionality to the best of our abilities.

Project Reflection Questions

1. **Would the Observer Pattern be useful in implementing the game board? Why or why not?**

No, there is no need to use Observer Pattern in using the game board. This is due to the fact that the board squares are not changing states much so they do not need to regularly notify other game board objects of the change of state. Second, players should not be automatically notified when one player's action causes the state of a building to change since it does not directly impact the game of any player, as that action can be bought to them when they land on that cell.

How could SLC and Needles Hall be modeled like Chance and Community Chest cards?

The **Iterator Pattern** is well-suited for managing Chance and Community cards. Since these cards are naturally organized as a stack(probability given),the iterator allows for an organized proper removal of cards. When the iterator reaches the last card in the stack, the iterator is reset to start from the first card again. Depending on the gameplay, shuffle or keep the deck same. This ensures a smooth and structured way to cycle through the cards while maintaining randomness.

2. **Is the Decorator Pattern a good choice for implementing property improvements?**

The Decorator Pattern is ideal in cases where objects need multiple levels of changes, such as a processor with various formatting. However, in this case, any changes just increase rent(tuition). Since there's only one way to change a building, there's no need for separate decorator classes with varied functionality. Instead, information such as the number of improvements and the respective tuition fees, can be retained within the building object itself by creating various variables.