**MUD Game Design Doc:**

Team Whatever

Matt Latura, Chris Conway, Michael Hogue, Mazen Shihab

This design doc features explanations of how each aspect of the MUD spec was implemented in our game and identifies each design pattern used in depth. Additionally we describe some of the decisions made during our development process and the extra credit features we have implemented.

Multiple clients can connect to the MUD game over the internet. To do so they should run the MainFrame class in the View package while the Server class is running either locally or remotely. The game will prompt them for the server IP, port number, user name and password.

Each client has their own player which they are able to control and move through the world. All players have names and locations within the game world. In the style of traditional RPGs, each player has stats which determine their effectiveness in combat including health points, strength, dexterity, precision, armor, and damage. Players will come across items in the game world which they will be able to pick up and store in their inventory. Examples of items include armor, weapons, keys, and even a mysterious sun wand. Each room in the world is placed on a 2d grid and the player can move between rooms through the use of commands detailed below. Players can chat, use social commands, and even attack each other if they desire.

The game world places the player outside of a mysterious dungeon and dares them to fight their way through its 30 rooms to kill the final dragon boss. Each room has a flavorful description including the room’s occupants, items, and exits. The combat system in our MUD is quite strategic. Players can advance and retreat to attack or avoid the attacks of enemy Mobs.

There are many types of items that can be found within the world. Weapons are one of the most common types and are used to boost a Player’s attack strength. Some weapons can only be used from melee range while others have a longer attack distance. Armor like items are quite commonly found on the dungeon floor or dropped from the enemy mobs. Armor gives the player a bonus to damage reduction in combat. 3 different items found throughout the game are able to interact with the world itself. Keys are used to unlock exits, Bombs can be used to reveal hidden items in certain rooms, and a mysterious sun wand item will temporarily bathe a room in sunlight which has the effect of warding off a certain pesky monster.

The mobs found within the dungeon are one of our games many highlights. Each of the 11 mob types has unique statistics, loot, and combat behavior. Some mobs in the game will attack each other on occasion while others may follow a running player through the dungeon.

Players are able to interact with the game world through the use of commands. These include movement commands such as North, South, East, and West. Look will print an updated description of the area the player occupies. Chat commands such as tell, say, and ooc allow players to communicate with the other players in the game. Inventory commands such as give, get, drop, and use allow the player to manipulate the items that he has.

The Mud takes place in a persistent game world. This means that players can safely log out and log back in to their characters without losing progress. Both items and mobs will re-spawn appropriately over time.

Following are descriptions of the different design patterns used in our MUD game as well as describing some of the decisions made during development and the extra credit features we implemented.

One of the design patterns we chose to use was the command pattern. Naturally, with a game that is playable by commands entered by clients on a server application, the command pattern was pretty useful. All of the commands that Clients could execute on the Server were placed in an enum. From there, we could determine if the first word that had been sent was contained in the CommandList enum. If the command was not in the enum, alerting the Client that their command was incorrect was very easy. If the command was there, we could easily determine which command to execute. We chose to use a switch statement over the CommandList to determine which Command to execute to make abbreviated Commands very easy to implement. A Command interface was developed, with an execute() method. All Commands that the Client could execute on the Server had a class that implemented this interface. After developing the commands, it was clear that three things were needed to execute all of the commands: the Server that the command needs to be executed on, the possible arguments/message that had need to be processed in a String, and the ClientManager that requested execution of the Command. Because of this, the Command interface implemented the execute method with those three parameters. The modularity of the project was much better as a whole, because if there was an error, we could narrow it down to the Command that caused it. The Server’s parsing of message had its readability and modularity increased, as well, because all complex actions did not need to happen inside of the parse method. Instead of every single command being processed in the servers parse method, we could simply write “new CommandName().execute(s, args, this);” and worry about processing the code somewhere else.

The Server contains many extra credit opportunities. One of the features is the abbreviated commands. Because we used an enum with a switch statement, it was very easy to use a ‘fallthrough’ design. For example, if the command was attack, there were possible abbreviations of att and a. With our design structure, we could just add ‘att’ and ‘a’ to the enum and place them above the attack case without a break, and they would enter the attack command.

Another extra credit opportunity we chose to include was server-side commands. Four buttons were added to the server, kick, shutdown, reset and ban by name. Kick will kick the player with the specified name out of the game, or alert that they are not there. Shutdown will save the game and exit the application (and is the default window close behavior). Reset will construct a new world with all new players and mobs where everything is back to new without having to close the server. Finally, ban by name will create a list of names that are not allowed to connect to the server. Social commands were another extra credit opportunity that we chose to implement .The five commands we chose were giggle, wave, dance, slap and highfive. By typing in one of those command and a character that is in the same room as you, all occupants were notified that you were, for example, waving at that character. One last server-side extra credit opportunity was implemented through alerting all players when somebody disconnects. Whenever there was an IOException, we could simply pass a message to sendToAllClients and they would all be notified that the player disconnected.

Initially, all the items in the GameModel had their own class; IronSword, Handaxe, etc., all extending Weapon, and the same for armor. This was ridiculously cumbersome, and each one didn't do anything but call the superconstructor. So, they were replaced using the factory pattern. Our ItemFactory takes a string which is the name of the item to one of 3 static methods: MakeConsumable, MakeArmor, or MakeWeapon. MakeArmor and MakeWeapon both call the armor or weapon subclass's superconstructors. MakeConsumable, on the other hand, makes instances of the various consumable subclasses; this is necessary, since each consumable has a unique use function. It also cannot generate sun wands, since these require a world passed to the constructor, but there's only one sun wand in the game, and it's just made using the SunWand constructor.

One of the extra features we did was the extra elements in the GUI, beyond the server/chat boxes. The minimap panel works similarly to the FaceUpPanel in the card demo from earlier in the year. First, it uses a MapSheet that has a BufferedImage containing every possible room layout in the game. The MapSheet has a method which will take a string containing the exits of the room wanted; for example, if it's a 3-exit room with the north wall blocked, you pass “ESW”. The method returns the 180x180 BufferedImage taken from the MapSheet which is the display for that room, and sets it to be

the image displayed on the MapPanel. We also have a JTabbedPane which contains 2 tabs: one for the user's stats, including health and statistics, damage stats, and experience. The other tab is a JScrollPane which will display every nonequipped item in the user's inventory. If the inventory grows bigger than what will fit in the inventory tab, then it'll get a vertical scroll bar, so that the user can always see everything they have in their inventory without having to use the inventory command and sorting through the output.

While designing the program we realized that one of the first things we would need to implement is a way to send information to the users based on what was happening in the game world, specifically to their player. The most common way to transmit this information in a MUD game is translating world actions into a text message and printing them to the appropriate player’s screens. Another common occurrence in our game specifically is updating the Player’s GUI including their mini map, statistics and inventory as the game state changes. To handle both situations we implemented an observer / observable relationship between the Server object and the World object. The Server object implements observer and adds itself to the Observable World object’s list of observers. Then, whenever the World has something it needs to tell a player in text form, or has to update one of the player’s GUI elements, it can call notifyObservers() and pass an object that contains any information it needs to send. Server’s update() method which receives these objects is able to determine what sort of information needs to be sent and it uses a number of ObjectOutputStreams to distribute it accordingly.

One complication with this system is that when the world has a text message to send to an individual or group of players it can’t simply send a String to the Server and expect the correct player to receive the message and his or her GUI to print it to the correct location. The solution we came up with for this obstacle was to wrap the String message in a Note class. This Note class adds functionality by keeping a Boolean value which determines whether this particular message should be printed to the chat text area or the game text area. The Note class itself is also wrapped by another class called Notification. This class adds even more functionality by keeping an Iterator of the Players who should receive the particular message. These two wrappers are examples of a Decorator pattern.

An example of the observer/observeable and decorator patterns in action is when a player sends the command to attack a mob. The World class receives the command and determines the outcome of the attack. Based on the outcome different messages will be generated. In this example let’s say that the World determines that the player has a hit his target (a giant rat) for 15 damage. The world will then create a Notification object which contains both a Note object and a player Iterator containing only the attacker Player. The Note object holds both the String message as well as a boolean indicating that this is not a chat message. The entire Notification is then sent to the server via a notifyObservers() call. When Server’s update() method recognizes the passed argument as a Notification it will proceed to iterate over the Players stored in the object. For each iteration it will locate that specific Player’s objectOutputStream and send the Note to the Player’s client. The client receives the note and knows based on the boolean that the message should be printed to the game text box where it’s user can now read “You have hit a Giant Rat for 15 damage.” The Notification system even gives us the flexibility to send a different message to different Players based on the same event happening in the World. The attacker, defender, and any other players in the room will all see different messages based on the same attack event.

When we wanted to implements the Mobs in the game, we decided to write a GameSimulation class that will create the different Mobs and will assign each mob a location and give the World of the game. The GameSimulation stores each Mob in an ArrayList of Mobs. This class was very helpful to our overall design of the game and the Mobs in particular. Upon request from the server, the GameSimulation will start the threads of each Mob. During the start of the threads, the simulation checks the status of the Mobs, avoiding starting dead ones. When a Mob dies, it sends a notification to the GameSimulation class, and the simulation remove the Mob from its list. There were many reasons to create this GameSimulation class. This class created a lot of flexibility for us. One of the main reasons we went with this design was to be able to save the status of the Mobs without saving their threads. In addition, we wanted to create a way to Re-spawn the Mobs and items periodically so this class was the perfect place to do so.