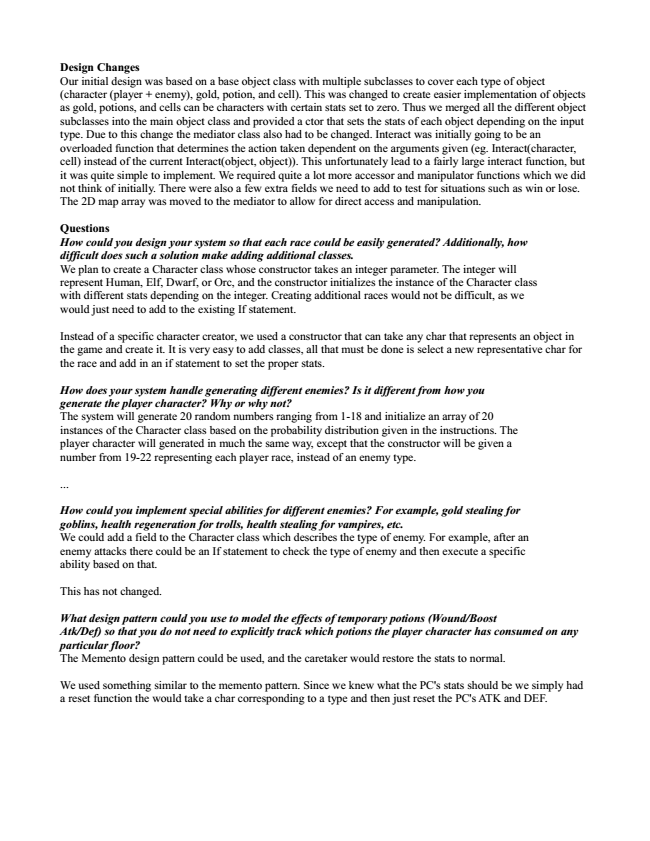


**Design Document**

Hunter Derkach & Matt White

Program Overview The design of the program uses two main classes to create the map for CC3K. The first class is Object and is used to create all the different objects that can appear on the map. It has fields for all the stats necessary for an object. The objects are put into sub categories using a subtype field. “player” for player controlled objects, “enemy” for npc objects, “gold” and “potion” for the respective items on the map, and “cell” for floor tiles, doors, walls, pathways, and the stairway. An object was created through a constructor which took in a char identifying the object (eg. 'h' for human or '9' for dragon horde, the numbers follow exactly from the assignment pdf), and an x and y position to identify its spot. The constructor takes the input and determines the correct data to be placed into the stat fields. With that, cells technically don't have stats, they are considered 0 stat objects, gold only contains gold, and potions have the appropriate stats for each potion type. Functions inside the object class consist of accessors and manipulators for each of the stats either adding or setting the stat directly.

The mediator class handles the map, a 2D array, and all the interactions between objects. It's constructor takes a file stream and reads in a map.txt to create the game board. It does this by reading in character by character assigning each spot in the array with an object corresponding to the map.txt file. After the initial map has been set it copies it into another array using a nested for loop to provide a base map for resetting the floor when the stairway is reached. The mediator also has contains a bool to keep the merchant's state of hostility. The main workhorses of the mediator are the attack function and interact function. The attack function was initially a part of interact but was separated to better handle combat for the player. Attack is called when the PC intends to attack a target. It determines if it is the appropriate target (eg. An enemy) or a cell, potion, or gold. It does this with if statements to determine the the correct subtype for each type of object. If it is not an enemy its does nothing. If it is an enemy it calculates damage and checks if HP has been reduced below zero. If HP is below zero it removes the object and sets the position equal to a floor tile object. If HP is below zero and the object is a merchant it sets the position equal to a merchant horde object. The next large function is interact. Interact handles all other need interactions with objects. Interactions implements PC & Gold object interaction, PC & Potion interactions, PC & Cell interactions (movement), Enemy & PC interaction (enemy attacks player, attack handles the reverse), Enemy & Gold, Enemy & Potion, Enemy & Cell. All interactions are determined by multiple if statements. Interaction covers special interactions such as elf potions usage, orc & dwarf gold collection, and PC dragon horde collection. These are all implemented using if statements and alter stats appropriately. For dragon horde pick up there is another function to determine if a dragon is near. Mediator's other main function is cycleEnemyActions() it uses a nested for loops to loop through the entire map and check each position for an enemy. When it finds an enemy it checks to see if it is in the radius of a player using nearPlayer(object &a). If this returns true the enemy attacks the player without moving. If this returns false, a random number is generated (seeded by time) and determines which direction to move. Interact(enemy object, object &b) is called to move the enemy appropriately. nextLevel() increments the level by setting map to cpyMap[][], which has the original map inside. It then resets the PC's ATK and DEF stat to its original amount, determined by its race. It then increments the floor. If the floor is ten it sets the win flag. SetDragons() loops through the map with a nested for loop and places a dragon in a position in the radius of a dragon horde on the map. It will check if the spot is a floor tile to ensure the dragon is not placed on a wall, pathway, stairway, character object, or door. The move<direction> functions call interact on the PC and an object in the radius of the PC in the desired direction. The attack<direction> functions do the same thing although it calls attack instead.



Design Changes Our initial design was based on a base object class with multiple subclasses to cover each type of object (character (player + enemy), gold, potion, and cell). This was changed to create easier implementation of objects as gold, potions, and cells can be characters with certain stats set to zero. Thus we merged all the different object subclasses into the main object class and provided a ctor that sets the stats of each object depending on the input type. Due to this change the mediator class also had to be changed. Interact was initially going to be an overloaded function that determines the action taken dependent on the arguments given (eg. Interact(character, cell) instead of the current Interact(object, object)). This unfortunately lead to a fairly large interact function, but it was quite simple to implement. We required quite a lot more accessor and manipulator functions which we did not think of initially. There were also a few extra fields we need to add to test for situations such as win or lose. The 2D map array was moved to the mediator to allow for direct access and manipulation.

Questions How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes. We plan to create a Character class whose constructor takes an integer parameter. The integer will represent Human, Elf, Dwarf, or Orc, and the constructor initializes the instance of the Character class with different stats depending on the integer. Creating additional races would not be difficult, as we would just need to add to the existing If statement.

Instead of a specific character creator, we used a constructor that can take any char that represents an object in the game and create it. It is very easy to add classes, all that must be done is select a new representative char for the race and add in an if statement to set the proper stats.

How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not? The system will generate 20 random numbers ranging from 1-18 and initialize an array of 20 instances of the Character class based on the probability distribution given in the instructions. The player character will generated in much the same way, except that the constructor will be given a number from 19-22 representing each player race, instead of an enemy type.

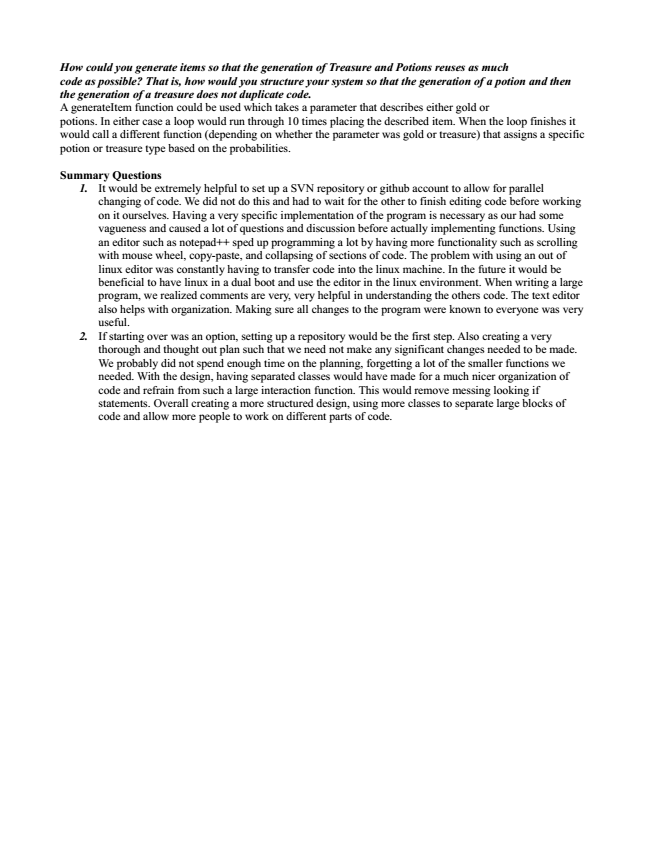
...

How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc. We could add a field to the Character class which describes the type of enemy. For example, after an enemy attacks there could be an If statement to check the type of enemy and then execute a specific ability based on that.

This has not changed.

***What design pattern could you use to model the effects of temporary potions (Wound/Boost Atk/Def) so that you do not need to explicitly track which potions the player character has consumed on any particular floor? The Memento design pattern could be used, and the caretaker would restore the stats to normal.***

We used something similar to the memento pattern. Since we knew what the PC's stats should be we simply had a reset function the would take a char corresponding to a type and then just reset the PC's ATK and DEF.



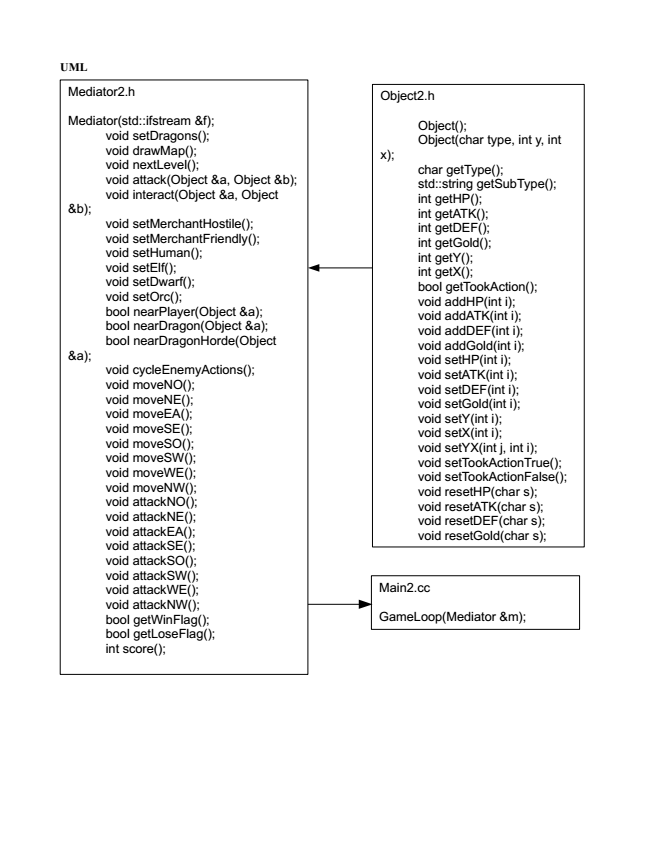
How could you generate items so that the generation of Treasure and Potions reuses as much code as possible? That is, how would you structure your system so that the generation of a potion and then the generation of a treasure does not duplicate code. A generateItem function could be used which takes a parameter that describes either gold or potions. In either case a loop would run through 10 times placing the described item. When the loop finishes it would call a different function (depending on whether the parameter was gold or treasure) that assigns a specific potion or treasure type based on the probabilities.

**Summary Questions**

1. It would be extremely helpful to set up a SVN repository or github account to allow for parallel

changing of code. We did not do this and had to wait for the other to finish editing code before working on it ourselves. Having a very specific implementation of the program is necessary as our had some vagueness and caused a lot of questions and discussion before actually implementing functions. Using an editor such as notepad++ sped up programming a lot by having more functionality such as scrolling with mouse wheel, copy-paste, and collapsing of sections of code. The problem with using an out of linux editor was constantly having to transfer code into the linux machine. In the future it would be beneficial to have linux in a dual boot and use the editor in the linux environment. When writing a large program, we realized comments are very, very helpful in understanding the others code. The text editor also helps with organization. Making sure all changes to the program were known to everyone was very useful. 2. If starting over was an option, setting up a repository would be the first step. Also creating a very

thorough and thought out plan such that we need not make any significant changes needed to be made. We probably did not spend enough time on the planning, forgetting a lot of the smaller functions we needed. With the design, having separated classes would have made for a much nicer organization of code and refrain from such a large interaction function. This would remove messing looking if statements. Overall creating a more structured design, using more classes to separate large blocks of code and allow more people to work on different parts of code.



**UML**

Mediator2.h

Mediator(std::ifstream &f);

void setDragons(); void drawMap(); void nextLevel(); void attack(Object &a, Object &b); void interact(Object &a, Object &b);

void setMerchantHostile(); void setMerchantFriendly(); void setHuman(); void setElf(); void setDwarf(); void setOrc(); bool nearPlayer(Object &a); bool nearDragon(Object &a); bool nearDragonHorde(Object &a);

void cycleEnemyActions(); void moveNO(); void moveNE(); void moveEA(); void moveSE(); void moveSO(); void moveSW(); void moveWE(); void moveNW(); void attackNO(); void attackNE(); void attackEA(); void attackSE(); void attackSO(); void attackSW(); void attackWE(); void attackNW(); bool getWinFlag(); bool getLoseFlag(); int score();

Object2.h

Object(); Object(char type, int y, int x);

char getType(); std::string getSubType(); int getHP(); int getATK(); int getDEF(); int getGold(); int getY(); int getX(); bool getTookAction(); void addHP(int i); void addATK(int i); void addDEF(int i); void addGold(int i); void setHP(int i); void setATK(int i); void setDEF(int i); void setGold(int i); void setY(int i); void setX(int i); void setYX(int j, int i); void setTookActionTrue(); void setTookActionFalse(); void resetHP(char s); void resetATK(char s); void resetDEF(char s); void resetGold(char s);

Main2.cc

GameLoop(Mediator &m);