**FINAL DESIGN DOCUMENT**

**By: Nerman Nicholas and Mohammad Sibtain Jafferi**

**DAY-TO-DAY PLAN (deviations from the original plan are noted and explained in the last column):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| DATE | GOAL | PERSON | Notes | Deviations |
| Friday, July 18th | UML | Nerman & Sibtain |  | N/A |
| Saturday, July 19th | Plan of Attack | Nerman & Sibtain |  | N/A |
| Sunday, July 20th | Entity/Floor | Sibtain / Nerman | Set up base class of everything (entity). Set up Floor so it takes a vector of entities  Floor has to be singleton | N/A |
| Monday, July 21st | Main | Sibtain / Nerman | Require a function to interoperate the map in main file. Must turn the map into a floor | N/A |
| Tuesday, July 22nd | Character Class | Sibtain / Nerman | Create the abstract character class that will set up all subsequent entities that move | N/A |
| Wednesday, July 23rd | Hero Classes / Enemy Classes | Sibtain / Nerman | Includes all child classes. Create all the separate abilities for enemies | N/A |
| Thursday, July 24th | Item Classes -> Potion and Treasure | Sibtain / Nerman | Potion must be an observer of player. Possible use visitor pattern to induce potion effect on player. | N/A |
| Friday, July 25th | Generation | Sibtain / Nerman | Generate all the above classes. | N/A |
| Saturday, July 26th | Graphics | Sibtain / Nerman | Work on setting up a graphic display of the game. | N/A – We did get this working but unfortunately due to segmentation faults and other errors that we were getting, we ended out DLC’s here |
| Sunday, July 27th | DLC | Sibtain / Nerman | Use images/sound/music in graphic display. Also have option to use keyboard movement within graphic display | We were not able to implement this DLC because we were still fixing bugs with the other classes for the actual cc3k specification |
| Monday, July 28th | DLC | Sibtain / Nerman | Random room generation. | We were not able to implement this DLC because we were still fixing bugs with the other classes for the actual cc3k specification |
| Tuesday, July 29th | DLC/Finish up/ Play the game | Sibtain / Nerman | Beta testing! Look for any possible bugs. | We tested the graphical display DLC thoroughly and we tested and fixed any remaining bugs with our program |
| Wednesday, July 30th | DLC/Finish up/ Play the game | Sibtain / Nerman | Beta testing! Look for any possible bugs. | We wrote up the final documentation on this final day |
| Days After | Practice for Presentation | Sibtain & Nerman |  |  |

**HOW WE IMPLEMENTED THE SPECIFICATIONS OF THIS PROJECT:**

Our overall structure of the program is as follows: (tabbed classes are children of the class above)

Entity

Character

Player

Human

Elf

Orc

Dwarf

Enemy

Vampire

Troll

Phoenix

Dragon

Merchant

Werewolf

Goblin

Item

Potion

potHP

potAtk

potDef

Treasure

Chamber

**Entity:**

An entity simply contains a character for it’s symbol on the board, an x and y integer and a string denoting the type of entity it is (“character”, “item”, “wall”, “empty”, “passageway”) to help determine which entities you can interact with. The remaining methods in entity are get and set methods that all children can use.

**Character:**

The Character class is used to sum up the similarities between the Player and Enemy class which are HP(int), Atk(int), Def(int), and race(string). Other methods include get and set methods for these private fields.

**Player Character:**

Player utilizes a singleton pattern. As outlined above the Human, Elf, Orc, and Dwarf classes are all children of the Player class. The Player class as a private fields attackedMerchant(boolean), actionMade(boolean), a pointer to the floor, a string called action, and a double called score. Score is basically the gold but if the race is equal to Human then score will be increased by 50%. Action is a string containing what the Player has done/seen in the current turn and this string is outputted to the user. The Boolean actionMade tells whether the player made an appropriate move according to the interpreter and if not then other entities in the floor will not act. The pointer to the floor is necessary in checking the surroundings of the player. The Boolean attackedMerchant is initially set false and set true and kept true whenever the player attacks a merchant. This field interacts with the Merchant class to determine whether all merchants are hostile to the player or not. For the protected fields I have a Boolean array of six elements that basically tells me if each of the potions have been seen so we can identify the potion when a player sees it. Gold is also a protected field as the subclasses take gold differently based on their racial features. The last three protected fields are bHP, bAtk, and bDef which hold the base attack, health, and defence stats so that I could refer to these when determining if a potion was applying an effect beyond the health of a player and etc. For public methods I have a move function, usePotion function, get and set methods for a bunch of private fields, an isDead function, a getScore function, and virtual overloaded methods usePotion() that take in a potAtk \* or a potDef \* or a potHP \*. There is also a virtual pickUpGold() used for races that pick up gold differently.

**Enemy Character:**

In the Enemy class I have protected methods hostility and moveable that are both Booleans and an integer called dropAmount. DropAmount is added to the player’s gold upon enemy death via the player’s pickUpGold method. Hostility is true by default for all classes except for merchants and dragons. checkHostility() is a virtual method that is useful in the merchant and dragon classes for determining if a player has attacked a merchant or is in a one block radius of the dragon horde (which makes hostility true). Only when hostility is true, then an enemy can attack a player. Other enemy methods include set and get methods for the private fields a move and attack method. The move method moves the enemy to a new valid position (using a method in the Floor class to see if the type of the entity at the random position the enemy wants to move to is moveable on). The move method only works if moveable is true so dragon is by default unable to move. Attack simply attacks the player and uses changeStats() on the player to modify its stats.

**Item:**

Item classes simply has a private Boolean field called available which informs us if the potion can be picked or not.

**Potion:**

In the Potion class we have a protected string field called potName used in printing the player action when he has already seen a certain type of potion. There are also protected Boolean fields called longEffect and used. Under private fields we have a pointer to the floor and Booleans available and positive (to indicate the different between RH and PH and etc). Methods in the potion class include effect() and its counterpart nullify() and get methods for the private fields. potAtk and potDef were done similarly but potHP was done a bit different. In potHP, longEffect will be true and there is another field called turns (integer) that takes away 1 point of health for 10 turns when the player uses a PH potion.

**Treasure:**

Simply has private field for the value(integer) of the treasure and a pointer to a player.

**Floor:**

Contains chambers, 2d array of all the entities on the map, win state determining fields and functions, utilities a singleton pattern. Has methods to draw player info after drawing the map with updated entity positions. Has methods to replace entities on the map with new entities that were made based on player actions. Has methods to determine if valid moves are made on the map.

**Chamber:**

Keeps track of enemies, potions, and treasure in the current chamber.

**PROJECT SPECIFICATION QUESTIONS:**

**Question:** 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?

*Due Date 1 Answer:*

Our system will design different races for a player as follows:

* Create a new sub-class for each race
* Have each race implement the virtual methods within the abstract class, Player
  + This includes: potionUse for now.
* This can be easily generated, as only a player pointer to specific subclass (race) is required
* Then this pointer can be used to attack/move/use potions through polymorphism.

The difficulty of adding new races with such a solution is limited to creating a new sub-class every time you require additional races. The virtual methods within this subclass must be properly defined. Hence, it is not extensively difficult to create not races/classes.

*Due Date 2 Answer:*

Our system generates different races based on the type of character. A Character has two child classes, Player and Enemy. The Player class contains child classes which are the specific playable classes. The Enemy class contains child classes of all the enemies. We use virtual methods in both the Player and Enemy classes to bring out the special features of different enemies and playable classes. Examples include like in the Player class we made usePotion a virtual method because an Elf has reversed effects when using a potion and for the other playable races they use the default usePotion() implemented in the Player implementation. We also have pickUpGold() in the Player class that is a virtual method because Orc’s and Dwarf’s pick up gold differently (either gold is doubled or halved). In our enemy class we use virtual methods for enemyDead() because certain enemies have different behaviour when they die. A Merchant drops a Merchant Horde and does not add gold to the player until the player walks over the hoard. A dragon just allows the player to take whatever is inside the dragon horde when it dies. We also have checkHostility() as a virtual method since merchants become hostile when one of their kind is attacked by the player. We implemented this by having a Boolean field in the player class that is initially false and turned true and left as true when the player first strikes an Merchant. The other enemy that needs to utilize checkHostility() is the Dragon class which simply checks if the player is in a one block radius of its dragon horde and if so then it will become hostile until the player leaves the one block radius. These races are easily generated as we only need a player/enemy pointer to specific subclass (race). Through polymorphism this pointer is used to attack/move/use potions according to each race’s features.

**Question:** How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

*Due Date 1 Answer [Nothing changed]:*

Our system will design different enemies as follows:

* Create a new sub-class for each enemy
* Have each enemy implement the virtual methods within the abstract class, Enemy
  + This includes: Attack and Ability for now

This is the same as the playable characters. This is because both the enemies and playable characters share similar patterns such as virtual methods like attack and move and other patterns such as all races having certain HP, Atk, Def. By having these similarities we created the Character class and followed a Factory Design Pattern for both heroes and enemies. This then made generating enemies and playable characters the same.

**Question:** How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

*Due Date 1 Answer [Nothing changed]:*

This will be done as follows:

* All different types of Enemies will have to define the virtual method “Attack” and “Ability”
* This Attack method will use a Player reference to change the player’s stats (e.g. deal damage)
* It will use the setter methods within Player to deduct the proper amount of stats.
* Then, if that Enemy’s attack induces an ability (e.g. life-steal), the attack method will call the ability method.
  + The ability method will change the Enemy’s stats by the appropriate amount, every time it’s called.
* However, if the Enemy’s ability is non-related to its attack, and is ongoing (e.g. HP regeneration), the ability will be called periodically by the Enemy class.

**Question:** 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?

*Due Date 1 Answer:*

We will use the observer pattern to model the effects of temporary potions. All potions will be observers of the player, and will be notified when the player leaves the chamber in which the potion exists. Each potion will have a boolean that specifies if it has been used. When a potion is notified, it will do 3 things.

1. Check if it has been used. If yes, then move on to 2, else stop.
2. Check if it is temporary. If yes, then move on to 3, else stop.
3. Disable the potion’s effects on the player.
   * This will be done by notifying the player of potion disuse

This way, you don’t need to explicitly keep track of any potion that has been consumed. All potions within a chamber are checked for temporary effects at the exit of a chamber.

*Due Date 2 Answer:*

In our Chamber class we have a vector of pointers to Potion objects which contains the Potions that are in a specific chamber. Using an aggregation relationship (empty diamond in the UML), we have an array of 5 Chambers inside the Floor class. When the Player leaves a floor we call the exitLevel method which goes through each chamber and runs a method that goes through each potion in the chamber and calls a function in the Potion class called stopUse() which is activated if the potion has been used and it’s still in effect (BA, WA, WD, BD).

**Question:** 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 generation of treasure does not duplicate code?

*Due Date 1 Answer [Nothing changed]:*

Our generation for items in a chamber will occur as follows:

1. Get the number of tiles in chamber
2. Use this to deduce the required number of potions and treasure in chamber (always = 10 potions and treasure/floor)
   * This will be done within a function that takes the chance of producing a certain cell, and a parameter that takes the number of cells. This reuses the code for determining not only the required number of potions and treasure, but also villains.
3. Create a vector of Item pointers
   * Item is an abstract class; the children of Item are potion and treasure.
4. Define the pointers to point to new potions and treasure.
   * Create the required number of potions and treasures
   * When creating new items, make sure each tile has an equal chance of obtaining the item. This will also be done through the previously stated function. Hence it will reuse code for potions, treasure and villains.
5. Use Floor to add these new Items to the chamber.
   * Entity is the base class for every tile in the game
   * Floor will have functions that adds Entities into the board
   * Hence, this reuses code because we will just need to loop through the vector and use the item pointers to get their position in the floor. Then, use the floor instance to access the Floor add method. This method will use the X,Y coordinates to add an Entity pointer of the specified potion or treasure to the floor.

This code only differs for potions and treasures in the adding segment. The rest of the code is reused for either item. This generation should be repeated for each chamber.

**FINAL QUESTIONS:**

**Question:** What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Developing software in teams brought many positive and negative aspects to getting the job done. For the positive aspects, having another individual with a different mind and different style of thinking can really help find the oddest of bugs. Having another partner also helps bring new ideas and even better methods of approaching a problem that you may not have thought of. The work was easily dividable amongst us and we were able to work on each-others parts through the use of proper documentation. We also learned a few downsides. One downside was coding style such as code name convention, indenting style, and overall spacing got a little annoying. Also having another partner in the project also meant having to deal with one partner being faster at doing one task and the other partner being faster at another task. The problem that arose with this is that each partner wanted to code different parts due to personal preference (in our case we both wanted to challenge ourselves and do what we felt least comfortable with). Writing large software requires **thorough planning** before jumping onto the code. We both failed to realize this and spent little time on the planning stage and figured out and modified the implementation and class definitions as we encountered issues with our implementation. We learned the importance of planning ahead as it helps give you a general idea of how everything interacts with each other and how you will go about implementing the various features in the project specification. Although thorough planning may involve using tedious techniques such as UML’s, it would have saved us a lot of time that we spent trying to fix segmentation faults and problems with how we implemented our program.

**Question:** What would you have done differently if you had the chance to start over?

If we have the chance to start over we would spend more than a couple of hours on the planning stage. We would read through each part of the project specification and discuss how we would go about doing it (taking down notes as we discuss this). After doing so, we would prepare a UML with good naming convention that both partners like (this would solve one of our problems when working together). After the UML was prepared then we would divide the tasks evenly. Since we have the proper code-structure figured out we know which classes interacts with what class and that way we can assign related classes to each other and not face the scenario where some parts of one class is implemented by one partner and the other parts by the other partner. This led to a very annoying path of debugging and modifying code implementation. As for our specific implementation, we would have loved to use enums to identify specific information the having to resort to using string ids. As for other features we did not like the use of keeping track of every potion and enemy in each chamber and checking the neighbours of the player every time he/she moves to update the action of seeing any nearby potions or enemies. We are not sure at this point of how we would implement this differently but we would definitely have a better idea if this was planned out better before we jumped into the code.