**Final Design**

Final Design Implementation

Implemented an observer pattern on the floor and the text display. The text display is notified every time something on the floor is spawned, killed, or used. All other types of floor tiles notify text display during generation of the map.

The Command Line Options we interpreted in a different way than what we later discovered was expected. Our interpretation allowed us to have map files that only contained a single floor, which specified the default starting value of the objects on the map. All other floors that were generated (including calling restart) would then generate a new randomized floor. We also implemented a method that completes the floor generation requirements if the given floor does not. This means that if the given floor only had 5 monsters, then we would generate the remaining 5. This is done while constructing a floor. We also use a default map so that all floors are generated this way.

This implementation led to an interesting situation with the Dragons and their Hoards. If a series of Dragons and DragonHoards were spawned, then an algorithm to sort through possible matches is created starting by taking all obvious cases (a Dragon that only has 1 adjacent Hoard) and removing them. Each time we remove a case, we restart the algorithm to see if there are now more possible matches. This allows us to narrow the possibilities for the Hoards. At the end, if there are still multiple cases remaining, a first-come-first-serve method is employed.

We built a hierarchy to allow us to create characters, items, or stairs so that tiles can have an occupant of any type. This means that we have a Thing class, which can either be a Stair, Character or Item. See the UML for the complete implementation of these classes. We first created a character which identifies the common elements between a PC and an Enemy. PC contains the methods that allow it to interact with items, such as collectGold and drinkPotion. A PC is also can be on the of the potential 4 races (Elf, Orc, Human, Dwarf). These classes simply manipulate the virtual method that is associated with its special ability (ie collectGold for Dwarfs and Orcs).

For Chambers, we a used a brute force method of creating a vector of Cells that then links together, creating a vector of 5 chambers. This is done to make it easy to spawn all things on the floor.

For movement, we created a singleton Game class which contains the current Cell of the player. When we are told to move the player, we remove the occupant from the player’s cell (ie the player) and then place it in the desired direction. This only works so long as the destination Cell does not impede movement. We also employ an Observer pattern to notify the Cells around the player that it has moved from its original location, and notify the neighbours of the new player cell that it has a player nearby. Enemies then strike if they are adjacent to the player’s new position.

Items can either be a Potion or a Gold. For Gold, we created a virtual getAmount function that returns the amount of gold depending on the type of gold. Normal gold (gold worth 1) is the base class for all other gold. When the player walks on gold, it called the PC method collectGold to notify TextDisplay that the gold has been obtained by the player. Potions are separated into two categories: basic and decorator. All potions use a virtual method applyEffect that takes in a PC. Basic potions manipulate a PC’s health through a setHP function from the player they are affecting. Decorator wraps the given attribute around the PC, and becomes the PC. Decorators are simply a PC containing a PC\* to the old PC. Thus, all Decorator methods call the methods for the PC, except for those that the potion manipulates. When the player uses a potion, the PC method drinkPotion, the player becomes the result of drinking a potion. The TextDisplay is then notified that the Potion has been used. This also updates a map contained in PC, which identifies the Potions the player has discovered.

Enemies are a Character, which are all hostile except for Dragons and Merchants. The hostility of an Enemy is tracked through a method isHostile. Their movement is also traced through the method canMove, which is true for all Enemies except for Dragons. Enemies can be of race Merchant, Werewolf, Troll, Goblin, Vampire, or Dragon. We have a set of Enemies in the Game class. We go through the set, which is properly ordered, and check to see if the player is nearby and the enemy is hostile. Then it will attack the player. If the player is not nearby, and it can move (ie there is a free space nearby and it’s not a Dragon) it will choose a random free space to move to and go to that location. Otherwise it stays where it is. Let it be noted, that Dragons will only attack the PC when it is nearby its hoard. When a Dragon dies, it notifies its gold that it can now be picked up the by the PC through the canPickUp method. This field in the DragonHoard is called the isMyDaddyDead.

Combat is done through a strike method that is a virtual from Character down. This means that when the PC kills an Enemy, then it can collect the bounty off the carcass. No gold is spawned on the map. Strike also calls the damage function which takes in the attacker’s attack and the and sets the defender’s HP accordingly. It does not need the defenders defence since the method is part of the defender.

Merchants will only attack the player if the field hasThePlayerHurtMyFriends. This is implemented through another Observer pattern which goes through the Game set of Enemies and notifies each Merchant that it should now attack the PC. This also notifies the Game field hostileMerchants to spawn hostile Merchants on all subsequent levels. Upon death, a MerchantHoard is spawned, notifying the TextDisplay that it has been created.

Stairs is simple class which is placed on the Floor in a separate chamber from the PC, unless otherwise specified. When the Stairs are stepped on by the PC, it calls Game’s method nextLevel, which loads the next level. If the player is on the last Floor, then Game’s gameOver method is called with value true (win). gameOver is also called with value false (lose) when a PC is killed. Restart clears the board, including the player, notifying the neighbours that the PlayerCell is changed. HostileMerchants are reset to default (false) and all Things are respawned. Then the games is started with a new Pick Race menu.

Generation is done in a multi-step process. Spawn is a Floor method that takes in a Thing\*. It picks a random place on the board that is not occupied by first selecting a chamber, then randomly selecting a tile in that chamber, making sure it is not occupied. It will then add the Thing\* as an occupant to the tile and return a pointer to that Cell, which used when storing enemies or the player cell. The only difference between each implementation will be the probability distributions specific to that type. This applies to Player, Stairs, Gold, Potions, and Enemies.

Note: our answers to the questions specified in cc3k.pdf have not changed since the Plan of Attack was submitted.

Questions

What lessons did this project teach you about developing software in teams?

This was not our first time working together in a project, so this project was more about recalling the fundamentals of cooperation over a significant period of time. Using source code sharing techniques is very challenging, especially when you are using a new program for the first time (like UNIX groups). When you run into problems with the software you are using, it often results in a large deviation of time from the project. In the initial set up for the code, it is often done through dividing up who creates what and how the file management/methods are going to be laid out. This means that each person has to read and learn each other’s code. This takes time out of the coding process because they have to spend more time understand the other person’s code than they do in actually writing their own code.

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

Our two primary classes (Game and Floor) have to high coupling, resulting in necessary overuse of “isType” virtual methods and static\_cast. Ideally, we would have like to have used a higher level design through Visitor and Observer patterns. We also would liked to have done some of the DLC material such as dynamic maps and arrow key movement. We also would have added more singleton classes on tiles cannot contain anything (empty space and the two types walls).