December 1st, 2013

Ruohan Chen (rhchen)
Alexander Fung (ac2fung)

ChamberCrawler3000 – Final Design

As stated in our original plan of attack, our project is still divided into four main classes. Firstly, there are the Player and Enemy classes, which are subclasses of the Character class, the Potion class, also a subclass of the Character class, and the Treasure class, a separate class. These classes provide an interface to get and set many of their fields as well as functions to interact with on another (i.e. attack each other, or use a potion, or pick up treasure). The Enemy class also has a Dragon subclass, the Potion class also had HPPotion and TempPotion subclasses, and the Treasure class also had a subclass for the dragon hoard. Then, we have our Floor class, which ties everything together and handles all the logic between how a user's commands in the test harness would lead to certain functions being called.

In terms of implementation, let's start with the layout. In the Floor class, we have an original layout, which contains the empty layout of the chambers, and a current layout, which is the same layout of the chambers, but populated with all items, enemies, the stair and the player. Every turn, we update the current layout, so that when coordinates of anything are changed, or if anything is deleted, the new layout would demonstrate that. In order to spawn, we made sure to have an array of two-dimensional arrays containing each chamber and their possible cells so that we can ensure choosing between the chambers randomly, and then choosing a cell in the chamber randomly. This also helps ensure that the player and the stairs spawn in different chambers. We then have array to pointers of all enemies, the treasure items, and the potion items (separately). This is helpful as when we update the current layout, we run through these three arrays and check their coordinates to update the floor grid. In this Floor class, we also have a Character pointer that points to the Player. The player is implemented with the singleton pattern so that there can only possibly be one player during the running of the game.

Regarding how the turns and interactions go along, let us start with moving the characters. When the user inputs a direction, the Floor class has a function that calculates the new coordinates, checks whether the new coordinates are a valid moving spot (i.e. an empty space, the passageway, the doorway to the passageway, or a gold hoard that does not have a dragon guarding it), and if so, sets the coordinates of the player character to the new location. When attacking an enemy or using a potion, the Floor class has a function that finds the character or potion in the direction that the user inputted. There then is another function that calls a method from the Enemy class to decrease their HP as they are being attacked, or a method from the Potion class to notify it is being used. If the potion is an HP potion, it then just calls a method from the Player class to increase its HP. If it's an enhancement potion, the potion acts as a decorator. That is, the main player pointer would then point to the enhancement potion instead (whose component is the actual player object), and the enhancement potion would still act exactly as the player object would (it provides the same interface). This is so that when the player moves on to a new floor, the decorators can just be deleted and the main player pointer can point back to the singleton instance of the player, so that the enhancements wear off. At the end of every turn, we run through the array of enemies and make them attack if the player is within a one-block radius of them. Otherwise, they move to one of the locations beside them randomly (after checking that it is an empty space, and hence a valid move for the enemy).

There are a few special features as well that required interesting implementations. Whenever a treasure was picked up, an HP potion was used, or an enemy died, they were de-allocated from memory, and their spot in their respective arrays would then become a null pointer. However, a dragon would first notify their respective dragon hoard when they died (essentially a modification to the observer pattern), so that the dragon hoard would know that it could be picked up when a user tries to walk over it. Otherwise, it would reject it and alert the user. With regards to the difference between whether a potion was seen or not, we implemented static boolean variables for each different type of potion in the potion subclasses, to indicate if they've been seen or not. When constructing the player and enemy objects, instead of having subclasses for each different possible player type and enemy type, we had an association list that mapped a char value to an array that contained all the specifications for each player or enemy type. Then the constructors would take in a char value and merely assign the specification fields (i.e. HP, attack, defense, etc.) by accessing this association list. This abstraction saves writing identical code. As well, by creating subclasses, we would have had to call virtual functions, which utilizes vtable, and hence would force the program to go through multiple pointers to fetch the correct function, incurring unnecessary overhead cost.

A few bonuses that we added – our custom layout file can also include the location of enemies, the player, and the stairs. This was done by running through the text file and identifying if a character was a number, if so, it was converted into the appropriate symbol for the object. Then we would add a corresponding object to its corresponding array (i.e. if there was a potion, we would make a new Potion that can be accessed through a pointer in the array containing all the potions). As well, these custom layouts can actually include more chambers if they wanted, in different layouts too, if we ever wanted to change up the map of the game a bit. Chamber recognition is actually done through a recursive call on neighbours of the first floor tile (i.e. '.') encountered, until all neighbouring chambers have been exhausted.

Questions (no major change of design):

Question 2.1

We designed our system such that our Player class (that is, the class that represents the player) is created differently depending on what the user inputs. Specifically, we have created an association list as a global variable that maps the types of the human player (represented by a character) to its associated HP, attack, defense, and gold multiplier values. The constructor would then access these values depending on what type was inputted by the user, and create the Player while setting its fields to said values. So, to add extra races, we would not even have to add extra classes – merely add more associations in the association list for new races, which is a simple process. Originally, we had considered creating subclasses for the player character. However, this would require adding an extra subclass for every new race, which requires more work and recompilation than our selected solution. By following what we have done, we have essentially abstracted all the differences away from the different possible player character types. In addition, we will be using the Singleton pattern for the Player class, such that there is only one player in the game at once. Please note that the Player class would also be the subclass of the Character class (to be explained in more detail in the next question).

Question 2.2.1

Our system generates different enemies in a somewhat similar manner to our Player actually. That is, the constructor would again take in the type of the enemy as a parameter (again represented by a character), and construct it by accessing an association list that maps the type of the enemy to its attributes again (that is, its HP, attack, defence, etc. values) to set the fields of the Enemy class to these values. It's done in a similar method, because considering the many types of enemies there are, this simplifies the code and prevents code duplication (it abstracts the differences away again), whereas this would be the case if we were to create many subclasses for each type of enemy. The Enemy class would be a subclass of the Character class, of which the Player class is also a subclass of, as they share many common attributes. Note that the Enemy class would not follow the Singleton pattern however, seeing that there are many enemies, and not just one, unlike the Player class. Furthermore, the Enemy type has an extra attribute that describes whether objects of the Enemy class are hostile, which helps to deal with the Merchant type of enemies. As well, another difference is that the Enemy *does* have a subclass. Specifically, the behaviour of the Dragon type is different from the other enemies – hence, there is a subclass for the Dragon type.

Question 2.2.2

In order to implement special abilities for different enemies, there are several ways this could be done. One possibility is to add a function to the Enemy class that would function differently depending on the type of the enemy using conditional statements (i.e. whether it's a merchant, goblin, etc.). It would then perform the necessary actions as required. Another possibility is to create different subclasses for each enemy similarly to the Dragon class – this would make sense as their behaviours would be modified with these extra abilities, and hence could deserve their own subclass. **Note: We did not implement extra special abilities other than those required since dragon did have its own class.**

Question 2.3.1

The decorator pattern could be used to model the effects of temporary potions. They could have accessor functions for the attack and defense that modify the return value on the accessor functions of their components. So, the original pointer to the single instance of the Player class would be preserved, and we could merely delete the decorator pointers when moving on to a new floor while setting the main character pointer back to the single instance of the Player class. This would help us modify the attack and defense values, without needing to keep track of how many potions are consumed on any floor.

Question 2.3.2

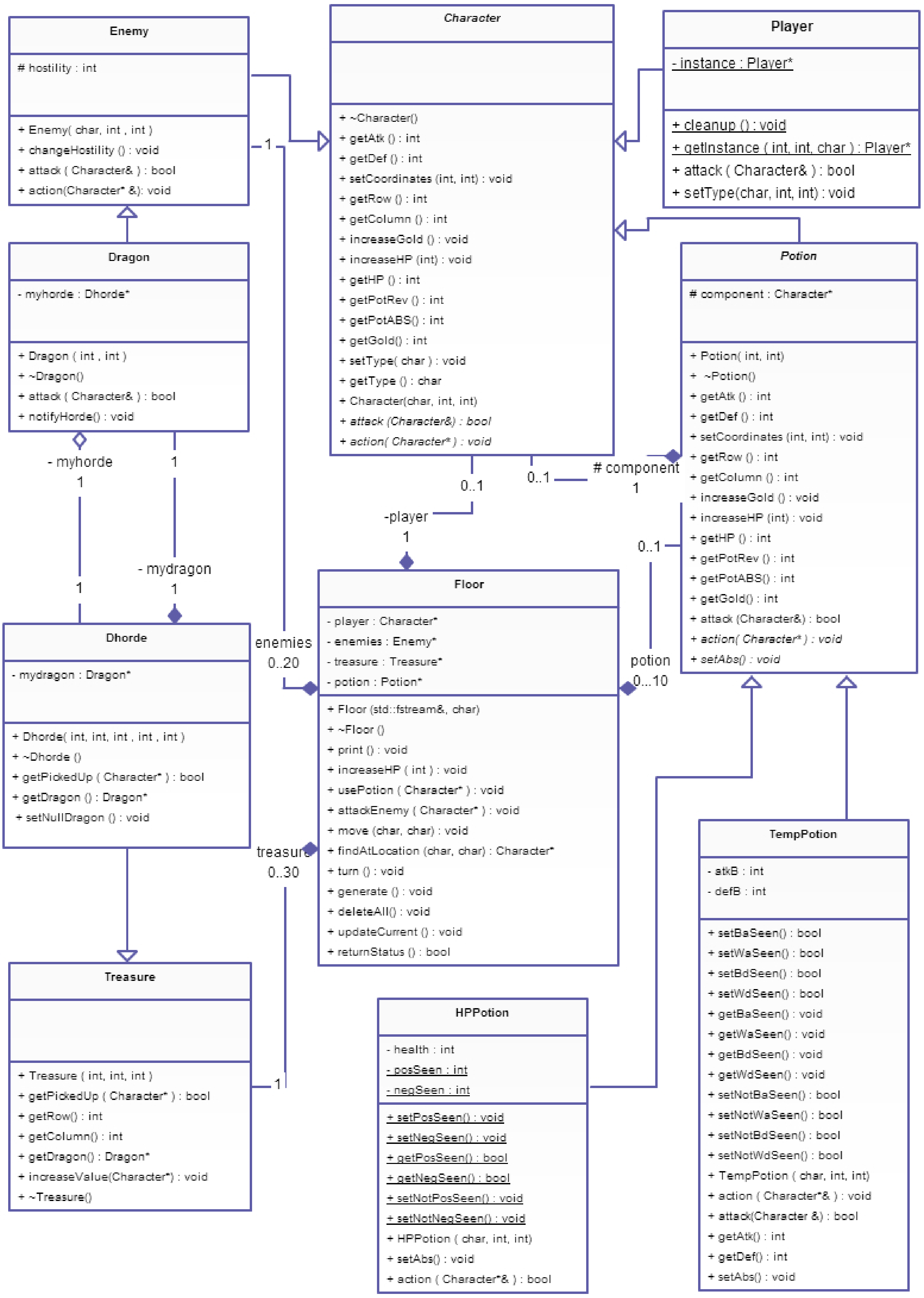
When generating the treasure, it would be helpful to not have too many different subclasses for the Treasure class. Instead of having different subclasses for different types of hoards of gold, it would be helpful to just have a field in the class that signifies the worth of the gold hoard. Note that we still need a subclass for the Dragon hoard as its behaviour is again different from the other types of hoards (this reflects how Dragon is also a subclass of the Enemy class). Similarly, for the Potion class, it only has two subclasses despite there being six types of classes. This is because the temporary potions are all essentially the same, and likewise with the HP potions. These abstractions help with preventing code duplications and reusing code.

Final Question 1

Developing software in teams is difficult. Communication is, as expected, remarkably important. Whenever any code is changed on your part, it is crucial to let your partner know what is happening, and why, so that everyone is on the same page at all times. Otherwise, precious time is wasted by updating your partner, or catching up. As well, it’s easiest to work in person together, as communicating about certain things virtually also has its obstructions.

Final Question 2

If we had a chance to start over, we would have spent more time thinking deeply about the design first, and attempting to foresee what obstructions would come up. Instead of diving into coding the program, it would be proved more beneficial to understand how any classes would work with each other, and how any aspects of the program could be simplified, or even if coupling could have been reduced through any means.


Enemy
# hostility : int
+ Enemy(char, int , int)
+ changeHostility () : void
+ attack (Character&) : bool
+ action(Character* &): void
Character
+ ~Character()
+ getAtk () : int
+ getDef () : int
+ setCoordinates (int, int) : void
+ getRow () : int
+ getColumn () : int
+ increaseGold () : void
+ increaseHP (int) : void
+ getHP () : int
+ getPotRev () : int
+ getPotABS() : int
+ getGold() : int
+ setType(char) : void
+ getType () : char
+ Character(char, int, int)
+ attack (Character&) : bool
+ action(Character*) : void
Player
- instance : Player*
+ cleanup () : void
+ getInstance (int, int, char) : Player*
+ attack (Character&) : bool
+ setType(char, int, int) : void
Dragon
- myhorde : Dhorde*
+ Dragon (int , int)
+ ~Dragon()
+ attack (Character&) : bool
+ notifyHorde() : void
Potion
# component : Character*
+ Potion(int, int)
+ ~Potion()
+ getAtk () : int
+ getDef () : int
+ setCoordinates (int, int) : void
+ getRow () : int
+ getColumn () : int
+ increaseGold () : void
+ increaseHP (int) : void
+ getHP () : int
+ getPotRev () : int
+ getPotABS() : int
+ getGold() : int
+ attack (Character&) : bool
+ action(Character*) : void
+ setAbs() : void
1
- myhorde
1
1
- mydragon
1
1
0..1
0..1
# component
1
-player
1
0..1
Floor
- player : Character*
- enemies : Enemy*
- treasure : Treasure*
- potion : Potion*
+ Floor (std::fstream&, char)
+ ~Floor ()
+ print () : void
+ increaseHP (int) : void
+ usePotion (Character*) : void
+ attackEnemy (Character*) : void
+ move (char, char) : void
+ findAtLocation (char, char) : Character*
+ turn () : void
+ generate () : void
+ deleteAll() : void
+ updateCurrent () : void
+ returnStatus () : bool
enemies
0..20
potion
0...10
treasure
0..30
Dhorde
- mydragon : Dragon*
+ Dhorde(int, int, int , int , int)
+ ~Dhorde ()
+ getPickedUp (Character*) : bool
+ getDragon () : Dragon*
+ setNullDragon () : void
TempPotion
- atkB : int
- defB : int
+ setBaSeen() : bool
+ setWaSeen() : bool
+ setBdSeen() : bool
+ setWdSeen() : bool
+ getBaSeen() : void
+ getWaSeen() : void
+ getBdSeen() : void
+ getWdSeen() : void
+ setNotBaSeen() : bool
+ setNotWaSeen() : bool
+ setNotBdSeen() : bool
+ setNotWdSeen() : bool
+ TempPotion (char, int, int)
+ action (Character*&) : void
+ attack(Character &) : bool
+ getAtk() : int
+ getDef() : int
+ setAbs() : void
Treasure
+ Treasure (int, int, int)
+ getPickedUp (Character*) : bool
+ getRow() : int
+ getColumn() : int
+ getDragon() : Dragon*
+ increaseValue(Character*) : void
+ ~Treasure()
1
HPPotion
- health : int
- posSeen : int
- negSeen : int
+ setPosSeen() : void
+ setNegSeen() : void
+ getPosSeen() : bool
+ getNegSeen() : bool
+ setNotPosSeen() : void
+ setNotNegSeen() : void
+ HPPotion (char, int, int)
+ setAbs() : void
+ action (Character*&) : bool