CS 246 Final Project - CC3k

Overview:

In this project of game named Chamber Crawler 3000, the inheritance feature of Object-Oriented language as well as different design patterns are utilized. The game allows player to choose different races with different abilities in combats with different enemies; the enemies also have different reactions which depend on the race the player has chosen. The game is played by the command line input, which is handled by main.cc and passed to the class Floor, then a sequence of methods will be invoked. The player is allowed to attack the enemies, pick potions, pick gold and move to any tile within the range of one block that is not occupied. If the player survives until reaching the stair of the 5th floor, the score will be calculated base on the gold that player collects throughout the journey. The design structure is modular. Each abstract class takes one specific responsibility or representing one kind of entity. More specific design strategies will be discussed below.

Design:

Spawn

`Floor` is the top class in our game that connects all the functionalities together and sets up the game; it owns class `Tile` by having a `vector<vector<Tile*>>` field that stores all Tiles and five `vector<Tile*>` that indicates which chamber a specific Tile belongs to. We are using Factory Pattern to spawn objects. Floor is the factory that is responsible for spawning. It will first choose a random chamber; next, a random empty Tile is chosen within that chamber, and the Tile's spawn method is invoked to generate different `Object` according to the given probability.

Random Generation

The way we spawn Object based on its own probability is to have arrays of names as a field inside of the class Floor. For example, we have an array of length 18 for the `Enemy` class. Inside this array, we have the 4 "H", 3 "W", 5 "L", 2 "E", 2 "O" and 2 "M", indicating the name of different kinds of Enemy. If we randomly choose one index of this array, it will give us the name of an Enemy based on the given probability. This strategy works in the same way for `Gold` and `Potion`.

Chamber of Stair and Player

In order to spawn `Stair` and `Player` in separate chambers, the Floor class records which chamber the player is spawned in its field `bornChamber`. The while loop of choosing a chamber for Stair can break if and only if the chosen chamber is different from `bornChamber`.

Combat

To avoid duplicate code, we used Visitor Pattern to implement combat between Enemy and Player. Because they all have their unique abilities, we override the virtual attack methods in all subclasses. The type will be identified through its pointer and the corresponding method will be invoked.
When we read in an attack command "a" with an attack direction "dir", `pAttack` method in Floor will be called. If the Object on the direction dir cannot be attacked, in which case it is not an enemy, we will ask the user to enter another command. If the Object is indeed an Enemy, the specific race of the player will call its `attackWho` method that is overridden from the abstract class Player. After the player finishes attacking, within the player's `attackWho` method, it will call the corresponding enemy's `attackBy` method. It is overridden from its abstract class Enemy; it wraps around the attackWho method. If this method is invoked, the Enemy will revenge back.

Legacy

attackWho of Player will also check if the enemy is dead, if it is, `Tile::objectDestroy()` will be called and the heap allocated Enemy can be deleted. Also, if the dead Enemy is `Merchant`, `Human` or `Dragon`, the relevant gold will be constructed or set to pickable.

Gold

The `Floor::pMoveTo()` method allows player to step on the intended Tile with `Small`, `Normal` Gold or `MerchantHoard`. Gold will be destroyed by its Tile, instead, Player will be the new Object on this Tile. If player wants to move to a `DragonHoard` while its guarding Dragon is still alive, this action will be treated as an invalid move and the player will be required to enter a new command.

Potion

When the user enters "u" command with "dir", this will call `pPotion` method in Floor. If there is no Potion in this direction, player will fail to pick potion and need to give a new command. We used Visitor Pattern in our abstract class `Potion`. If there is a Potion on the intended Tile, we will first get the name of the Potion and through the Player's `pickPotion` method to call `usePotion` method overridden in the corresponding concrete subclasses of this type of Potion. If the player has never picked this Potion before, the Potion's name will be added to the player's `knownPotion` field, which is a `vector<string>`.

Move

Player Move

Since Floor is the mode of the entire program, so we simply pass the direction string to the Floor method `pMove(dir)`. It wraps the `playerMove(dir)` method in the class `Player`. It also determines if the Tile is steppable, if it is, `pMove(dir)` will invoke `playerMove(intend, dir)` and returns true and false otherwise, so we can save the returned boolean in main.cc to determine if we need to ask the user for an additional valid move command. The `playerMove(intend, dir)` takes the single responsibility of moving the player to Tile intend. When the move is completed, the method `objectEmpty()` of Player's tile field will be called to clean the selected Tile without destructing the Object on it (Player).

Enemy Move

Different from how Player moves(which is controlled by user), Enemy moves whenever the Player completes a valid action (attack, move, or use potion). The class Floor still takes the duty of moving the Enemy; however, since Enemy should move randomly after the player acts, we call `Floor::moveEnemy(enemy)` before "return true;" for of `Floor::pAttack(dir)`, `Floor::pMoveTo(dir)` and `Floor::pPotion(dir)`. Floor will loop through all of its Tiles to see if it has an Enemy; if it does, Floor will invoke the `enemyMove(p, ifFreeze)` method in the class `Enemy`.
Moreover, `enemyMove(p, ifFreeze)` first checks if the Player p is near to the Enemy itself to determine whether to stop and attack the Player or not. If it does not attack the Player and it is not freezed, it will move to a random empty Tile (`Space`) within one block range. Similar to the Player's movement, if the Enemy moves successfully, objectEmpty() of Enemy's tile field will be invoked.

Move Restriction

In terms of the move restrictions, since a player can step on `Passage`, `Door`, `Gold` and empty `Tile`, we initially implemented a virtual method in the class `Tile` and override it in Tile's subclasses. Nevertheless, after considering Enemy has different move restrictions compared to Player, we ended up checking if the symbol that represents a specific Tile instance satisfies the restrictions and whether the Tile is empty (`Gold` is an exception as explained in above section). If both conditions are satisfied, the Tile will be considered as steppable.

Text Display

We use Observer Pattern to in order to change the game map.
The `TextDisplay` is the observer of each individual Tile of the Floor, whenever a Tile changes its state, which happens when there is Object moving or vanishing, the `TextDisplay` will be notified and updated automatically. More specifically, when `Tile::objectEmpty()` or `Tile::objectDestroy()` is called, the tile will notify

the `TextDisplay` and the `TextDisplay` will update this cell. In addition, when the Player or the Enemy has any new action, this action will be added to the `TextDisplay` by `TextDisplay::addAction(act)` or `TextDisplay::cleanAction()`. Considering that `TextDisplay` does not have to manipulate any data in other classes and its only responsibility is to store the map of the game, the coupling can be loosed in this situation.

Questions Revised:

Player Character

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 races?

Answer: In our design, we have an abstract base class `Player` that has all races as its concrete subclasses. `Player` contains pure virtual method `attackWho` which can be overridden in every concrete subclasses. In `Player` class, we implemented `useGold` and `pickPotion` methods for subclasses to inherent from.
This design pattern allows us to add new races by just adding concrete subclasses that override the `attackWho` method with respect to their own feature.
(We changed our `attackWho` method from virtual to pure virtual since every races of Player Character has different effect on different enemy races.)

Enemies

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

Answer: It is different from generating the player character. We will spawn our player character at the beginning, followed by stair spawn. Factory pattern is used to generate multiple instances of enemies. The race of the player character is chosen by the user before spawning; however, when the enemies are spawned, the user does not have control of it. Instead, the class `Floor` is the factory that spawns the enemies randomly.

Question: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

Answer: We used Visitor Pattern for both player and enemy characters. Under the abstract class `Character`, we constructed another abstract class `Enemy` that all enemy characters will be inherited from. The pure virtual method `attackWho` is overridden by different enemy races, depending on their opponents in a combat and what effects they have correspondingly to the player.

Items/Potions

Question: The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by weighing the advantages/disadvantages of the two patterns.

Answer: At first, we decided to use Decorator Pattern on different potions. However, when we implemented the abstract class `Potion`, we noticed that we need to consider every race of enemies' maximum health point and their unique abilities when they picked up a potion. Thus, we changed to Strategy Pattern which allowed us to apply `usePotion` method interchangeably on different player races for different effects.

Items/Treasure

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?

Answer: We used Factory Pattern to generate `Potion` and `Gold`. As `Potion` and `Gold` belongs to their abstract base class `Object`. We generate potions and gold after the player and stair are spawned. In `spawnGold` and `spawnPotion` methods in floor, we randomly selected one type of gold/potion which names are stored in an array of `goldName/potionName`, a chamber and an empty tile within that chamber in order to "new" a gold/potion as the object on that tile. As a result, we are able to generate required number of potions and gold in one for loop.

Final Questions:

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?

By doing this project, we learned that it is important to exchange ideas with each other, engage every group member, divide tasks among the group clearly and design the structure well before we actually start to code. By designing the program well and understanding the specifications of the program, programmers are able to avoid spending time writing unnecessary code and and debugging. Moreover, it allows us to write compact and efficient program and thus significantly improve the quality of our program, which is very important in our future career. In addition, it will improve the productivity if the task that assigned to one group member is more of an entire class with its abstract base class and all concrete subclasses. Since the coupling between

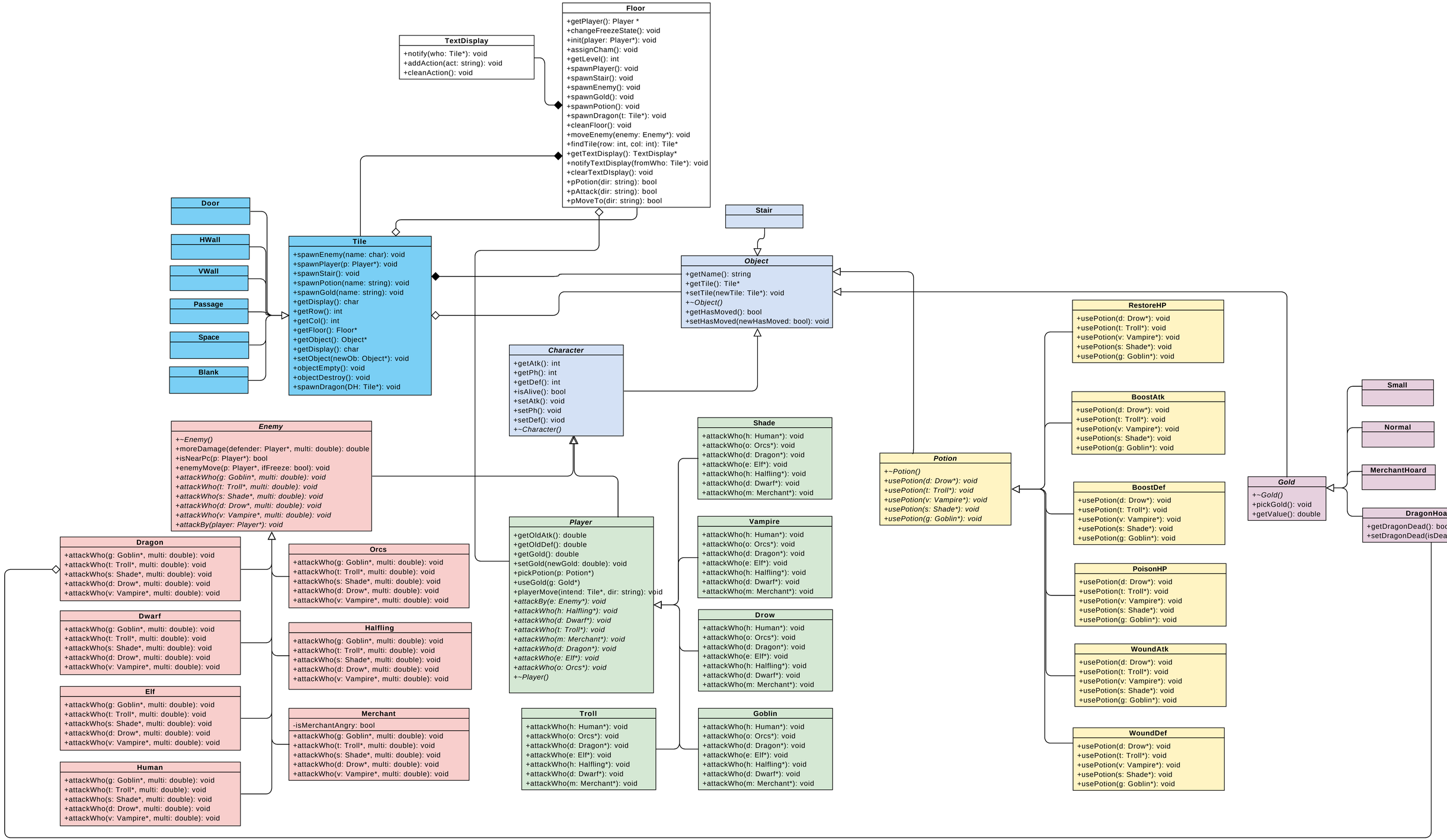
these different upper classes are relatively, it will be less possible that duplication or different naming style happens.

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

If we had the chance to start over, we would have spent more time designing the structure of the program and discuss with group members to make sure we understand each other's idea. By doing so we can significantly reduce the time we spent in writing unnecessary code, trying to understand other member's code or debugging.

Summary:

Our implementation of CC3k uses the MVC framework by separating mode module, view module and controller module of the game. Our model is the `Floor` class which is directly related to all the game components including player, enemies, potion, gold, chamber and tiles. The view of the game is the `TextDisplay` class and its sole responsibility is to display the game board on the terminal screen every time the state of the game changes. The controller is in `main.cc`, which takes the responsibility of interacting with the user and processing the commands from the user. The change propagation mechanism functions due to the link between mode and view. In terms of design patterns, class `Gold, Potion, Enemy` and `Player` use Visitor Pattern, Factory Pattern and Strategy Pattern as different combinations may invoke different abilities. This allows us to apply a set of algorithms to achieve the same goal for different character types regarding their various abilities which enhance our cohesion of our program. Since we implemented subclasses for `Character,Gold,` and `Potion` and overrode the virtual methods in the concrete subclasses, this minimise the coupling between different object types. Finally, in class `Tile`, we used the Observer Pattern, whenever there is a change on the game board, it will notify its own observer, `TextDisplay`, to display the latest change on the screen. In conclusion, this project provides us a good opportunity to practice what we've learned throughout the term and to develop a deep understanding of object-oriented programming.


Floor
+getPlayer(): Player *
+changeFreezeState(): void
+init(player: Player*): void
+assignCham(): void
+getLevel(): int
+spawnPlayer(): void
+spawnStair(): void
+spawnEnemy(): void
+spawnGold(): void
+spawnPotion(): void
+spawnDragon(t: Tile*): void
+cleanFloor(): void
+moveEnemy(enemy: Enemy*): void
+findTile(row: int, col: int): Tile*
+getTextDisplay(): TextDisplay*
+notifyTextDisplay(fromWho: Tile*): void
+clearTextDIsplay(): void
+pPotion(dir: string): bool
+pAttack(dir: string): bool
+pMoveTo(dir: string): bool
TextDisplay
+notify(who: Tile*): void
+addAction(act: string): void
+cleanAction(): void
Door
HWall
VWall
Passage
Space
Blank
Tile
+spawnEnemy(name: char): void
+spawnPlayer(p: Player*): void
+spawnStair(): void
+spawnPotion(name: string): void
+spawnGold(name: string): void
+getDisplay(): char
+getRow(): int
+getCol(): int
+getFloor(): Floor*
+getObject(): Object*
+getDisplay(): char
+setObject(newOb: Object*): void
+objectEmpty(): void
+objectDestroy(): void
+spawnDragon(DH: Tile*): void
Stair
Object
+getName(): string
+getTile(): Tile*
+setTile(newTile: Tile*): void
+~Object()
+getHasMoved(): bool
+setHasMoved(newHasMoved: bool): void
Character
+getAtk(): int
+getPh(): int
+getDef(): int
+isAlive(): bool
+setAtk(): void
+setPh(): void
+setDef(): viod
+~Character()
Enemy
+~Enemy()
+moreDamage(defender: Player*, multi: double): double
+isNearPc(p: Player*): bool
+enemyMove(p: Player*, ifFreeze: bool): void
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
+attackBy(player: Player*): void
Player
+getOldAtk(): double
+getOldDef(): double
+getGold(): double
+setGold(newGold: double): void
+pickPotion(p: Potion*)
+useGold(g: Gold*)
+playerMove(intend: Tile*, dir: string): void
+attackBy(e: Enemy*): void
+attackWho(h: Halfling*): void
+attackWho(d: Dwarf*): void
+attackWho(t: Troll*): void
+attackWho(m: Merchant*): void
+attackWho(d: Dragon*): void
+attackWho(e: Elf*): void
+attackWho(o: Orcs*): void
+~Player()
Shade
+attackWho(h: Human*): void
+attackWho(o: Orcs*): void
+attackWho(d: Dragon*): void
+attackWho(e: Elf*): void
+attackWho(h: Halfling*): void
+attackWho(d: Dwarf*): void
+attackWho(m: Merchant*): void
Vampire
+attackWho(h: Human*): void
+attackWho(o: Orcs*): void
+attackWho(d: Dragon*): void
+attackWho(e: Elf*): void
+attackWho(h: Halfling*): void
+attackWho(d: Dwarf*): void
+attackWho(m: Merchant*): void
Drow
+attackWho(h: Human*): void
+attackWho(o: Orcs*): void
+attackWho(d: Dragon*): void
+attackWho(e: Elf*): void
+attackWho(h: Halfling*): void
+attackWho(d: Dwarf*): void
+attackWho(m: Merchant*): void
Troll
+attackWho(h: Human*): void
+attackWho(o: Orcs*): void
+attackWho(d: Dragon*): void
+attackWho(e: Elf*): void
+attackWho(h: Halfling*): void
+attackWho(d: Dwarf*): void
+attackWho(m: Merchant*): void
Goblin
+attackWho(h: Human*): void
+attackWho(o: Orcs*): void
+attackWho(d: Dragon*): void
+attackWho(e: Elf*): void
+attackWho(h: Halfling*): void
+attackWho(d: Dwarf*): void
+attackWho(m: Merchant*): void
Dragon
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Dwarf
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Elf
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Human
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Orcs
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Halfling
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Merchant
-isMerchantAngry: bool
+attackWho(g: Goblin*, multi: double): void
+attackWho(t: Troll*, multi: double): void
+attackWho(s: Shade*, multi: double): void
+attackWho(d: Drow*, multi: double): void
+attackWho(v: Vampire*, multi: double): void
Potion
+~Potion()
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
RestoreHP
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
BoostAtk
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
BoostDef
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
PoisonHP
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
WoundAtk
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
WoundDef
+usePotion(d: Drow*): void
+usePotion(t: Troll*): void
+usePotion(v: Vampire*): void
+usePotion(s: Shade*): void
+usePotion(g: Goblin*): void
Gold
+~Gold()
+pickGold(): void
+getValue(): double
Small
Normal
MerchantHoard
DragonHoard
+getDragonDead(): bool
+setDragonDead(isDead: bool): void