Leighton Chen & Dave Li
lzchen & dzli
CS246: CC3K Final Project
Final Design Document

1 Design

The basis of our program are the Game, Floors, and Cells. Floor is an abstract class, and it is the superclass Level and Chamber. Each Game has 8 Levels, and each Level has 5 Chambers. Each chamber consists of varying number of Cells, and there is a wide variety of different kinds of Cells.

1.1 Floors: Chambers and Levels

Lets start with the core. As aforementioned, Floor is an abstract class with methods, tick(), spawn(), and print(). Chamber inherits these from Floor. Chamber has two arrays as fields: one is an array of pointers to Cells called Layout, the other is an array of chars called Display. When we construct a Chamber, we have to initialize these two arrays. For Layout, we create the array and initialize it with floor tiles. Since Chambers 2 and 5 are not rectangles, we hard-coded the construction of these two chambers by setting the empty cells to NULL. Then, we use the function updateDisplay() that takes the Layout, and as you may have guessed, updates the Display by iterating through the Layout and places the character corresponding to the Cell in the Display. Chamber's print() prints out the Display, tick() moves all the enemies in the Chamber (as well as attack the Player if present), and then calls updateDisplay().

Level also has an array as a field, but only one and it's Display. However, it also has a pointer to the Player as a field, as well as 5 chambers. It's a bit more complicated, as the player's movement is handled here. We chose to do this because Players can move outside of Chambers, so working with movement in Chambers wouldn't be possible. Instead, we move based off the the Display which is more memory efficient, and if the Player interacts with a Cell, we would access the chamber the Player is currently in (as all chambers are fields of Level) and interact with it there. A key point is that Level doesn't have a Layout field, which is different from what we planned initially. We chose not to include a Layout of Cells because most of Levels aren't Cells: it would be a waste of memory and inefficient to include an array of mostly NULL pointers.

We implement spawning by using rand() in <cstdlib>. Whenever we want to spawn anything other than a player, we call rand() three times: once to determine which chamber, as each chamber has equal probability of getting an object despite being different sizes; once more to to determine where in the chamber to put the the object, recalling rand() if it's an invalid place to put it (if it's a NULL Cell or if it's occupied); and once more to determine what kind of the object to spawn.

1.2.1 Cells: Characters, Enemies, and the Player

The whole game is centered around the characters and interactions between characters. We chose to create an abstract Character class that is never constructed, with subclasses Player and Enemy. Enemy has a subclass for each Enemy type, be it werewolves, vampires, etc. Most of the enemies are unremarkable: they move around randomly, attacking the player whenever in range, and the only distinction between any of them are their base stats and spawning chance (as a guideline, the easier the opponent, the higher the spawn rate). Two enemies stand out: the merchant and the dragon. Merchants are neutral enemies that only become hostile after the Player attacks one. We decided to implement this with a static boolean called merchantHostile. Before the merchant would attack, it would check whether or not this variable is true, and it would only attack if it was. And since it is a static field, all Merchants know whether or not to attack the Player. Dragons on the other hand do not move, but instead are around its dragon horde (called DragonGold in our code). Dragon has an overloaded moveRandom() that prevents it from moving.

The Player is the sun of that solar system that is this game: that is, everything revolves around it. It can be one of four races: Humans, Elves, Dwarves, and Orcs. Humans get a bonus 50% to their score at the end, Elves get the absolute value of any potion effect (more on potions later), Dwarves have lower base stats but get double the gold, while Orcs have higher base stats but get half the gold. We implemented each race as a subclass of a Player class. Player also has special dATK and dDEF fields, which are used for potions. For these special abilities of races, we overloaded functions that are in player. For Dwarves and Orcs, we made getGold() return double and half the amount of gold respectively. Elves get a new usePotion(Potion* p) function that takes the absolute value of the effect of the potion.

We were discussing the possibility of using a Singleton design for Player, but we decided not to because we create multiple instances of the Player when moving from chamber to passage, as the passage isn't a cell since Level doesn't have a Layout.

1.2.2 Cells: Items, Potions, and Gold

Items are objects that enhance (or impair) the Player's experience in these levels. Item is our abstract class that is the superclass for Potions and Gold.

Potions are used by the Player to boost stats. At first, we were thinking about using the Decorator pattern to implement potions, but we came up with the idea of just having two fields in Player (dATK and dDEF) that Player's usePotion(Potion* p) can modify. We decided to do this because it is much more simple than implementing a decorator. usePotion would get the Potion's type and magnitude (both of which are fields of potions) and apply them to the Player by modifying dATK and dDEF. When the Player calls getATK() or getDEF(), it returns (ATK + dATK) or (DEF + dDEF) respectively. That way, we always know what the Player's base stats are (ATK and DEF), for when Players enter new floors, these two are reset back to base. For that, we have a resetStats() function for Players as well, to be called when a Player enters a new floor.

Potions also have other interesting behaviors. When an Player sees a potion, at first, it is unknown what type it is to him, but after using it, it reveals itself for all subsequent uses of that potion. We did this the same way as we did Merchants: a static bool for each kind of potion and whether it's been discovered before, as well as a static function to change it the first time it is discovered.

Gold is all roughly the same, with the exception of Dragon Hordes (called DragonGold in our code). It spawns randomly throughout the level, in amounts of 1 or 2. Because of this, we decided to make the gold a singular class with an amount field instead of separate classes. This has the exception of DragonGold, which has a pointer to a Dragon. In our spawn() function the Dragon is placed whenever a DragonGold is spawned. Also, whenever the Player tries to move on the DragonGold to collect it and the Dragon is alive, nothing happens. This is handled by Level, as that takes care of Player's move.

1.3 Game and Main

Everything comes together in Game and Main. Game initializes a Level and its Chambers and Cells, and then calls tick whenever the Player enters a command in the Main. Game also has a field which is the Pointer to the Player. Game also prints the layout from Level, along with the Player's stats. The pointer to the Player serves so that we can always get the Player's stats (especially X and Y position), and so that we can store it when it goes out of chambers.

Main takes in input from stdin and matches it with a command that we discussed before. After every action, Game calls tick() which calls Level's tick(), which calls Chamber's tick().

2 Design 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 designed our system so that each race was a different class, all of which inherited from an abstract Player class. This has not changed from our planning stage. This inherited Player class had virtual methods such as getATK(), getDEF(), and other general methods every race needed. However, for the separate races that had special abilities, we simply overloaded methods that were already in Player. For example, as Dwarves got double the gold, we had Dwarves' getGold() override Player's and it would return two times the gold instead of Player's one times. We believe this implementation was efficient as we didn't have to duplicate most of the code, just for methods that were unique for a certain class. A good example of this would be for Elves, who never had any negative potion effects. To implement the elves' special usePotion() method, we simply copied over Player's and just edited so that changeATK(), changeDEF(), and changeHP() took the absolute value of the intended effect of the potion.

We believe our solution to this problem made it extremely easy to create additional classes. If we wanted to make a new class, say an Alien class, we would create a new alien.cc, that would inherit from player. We would write any new functions needed (maybe it could absorb the stats of enemies he killed? We would just rewrite attackEnemy() in that case.), and simply include the Alien in our main function when asking for the player type.

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

Our system handles generating different enemies by using Level's generate() function. In this function, we create all the chambers, and in doing so, spawning the enemies. We call rand() once to determine which chamber to put the enemy in. Then, we call rand() one more time to determine which cell in that chamber to put it on, recalling rand() if that cell is a NULL cell (like the ones in the two rightmost chambers) or if it is already occupied, checked by the display of the cell. Then, we call it a final third time to determine which monster to actually spawn, with their weighted probabilities. Our implementation has not changed from our planning stage.

This is almost exactly the same to how Player's spawn. Since we decided that Players and Enemies were both Characters and could inherit methods from Character, it was a matter of simply waiting for input on stdin before we could spawn the Player. So instead of calling rand() a third time, we wait for input on stdin, and check what that is, and spawn the corresponding race.

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

As we implemented each enemy as a different class under a broad Enemy class, implementing special abilities for different enemies would be simple. We could just overload the functions in that particular race while reusing code from Enemy. For example, if goblins were allowed to steal gold, then we would copy the attackPlayer(Player * p) function from Enemy, and in that decrease p's gold and increase the goblin's gold at the end of every attack.

> 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 has consumed on any particular floor?

We could have used a Decorator design pattern to model the effects of temporary potions, with Potion classes decorating the Player, and adjusting their stats. However, we thought of a much more simple method: simply add two fields to the Player, called dATK and dDEF (for change in ATK, and change in DEF). getATK() and getDEF() would return the sum of the stat and the change in that stat. This way, we do not need to explicitly track which potions the player has consumed as these two fields do it for us. An added bonus to this method is that resetting to the base ATK and DEF when moving to a new floor is simple. We would just reset these two fields back to 0 whenever the player would go onto a new floor.

> 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 the generation of treasure does not duplicate code?

We decided to separate Treasure and Potion into different subclasses of the superclass Item. When we make a new level, it calls the constructors of Treasures and Potions, and when we place them, we do so through calling rand() three times in a for loop. However, we did not generate these together as we planned, mainly because it was more logical to have them separate. We did duplicate some code, but I think it made our code much easier to follow to have separate sections to generate potions, and treasure, and enemies.

3 Project Questions

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

This project taught us that although software development in teams lessens the amount of work each has to do individually, it raises problems in coordination and working together. Thankfully, the two of us worked together pretty well, but lack of communication and documentation of changes we made did get frustrating at times. For example, there would be times we would both work on the same .cc file, and be unable to save both without writing over one of them. This was easily avoidable, but it took a couple of attempts to get in the habit of notifying the other when we would work on a certain file.

Another issue we had was documenting what changes we made. A problem that arose because of this was the way chambers were implemented: we set the first coordinate to be the Y, while the second was the X. Dave didn't realize this was our implementation, and only after writing a decent amount of code that caused lots of segmentation errors did he figure this out, and he had to go back and change all that code he made. His time could have been spent more efficiently by working on something else.

The major lessons this project taught us would be to combat these two issues: communication and documentation. They are both essential to good teamwork for developing software, so that a member of the team knows what to work on, what not to work on, and to understand what someone else did.What would you have done differently if you had the chance to start over?

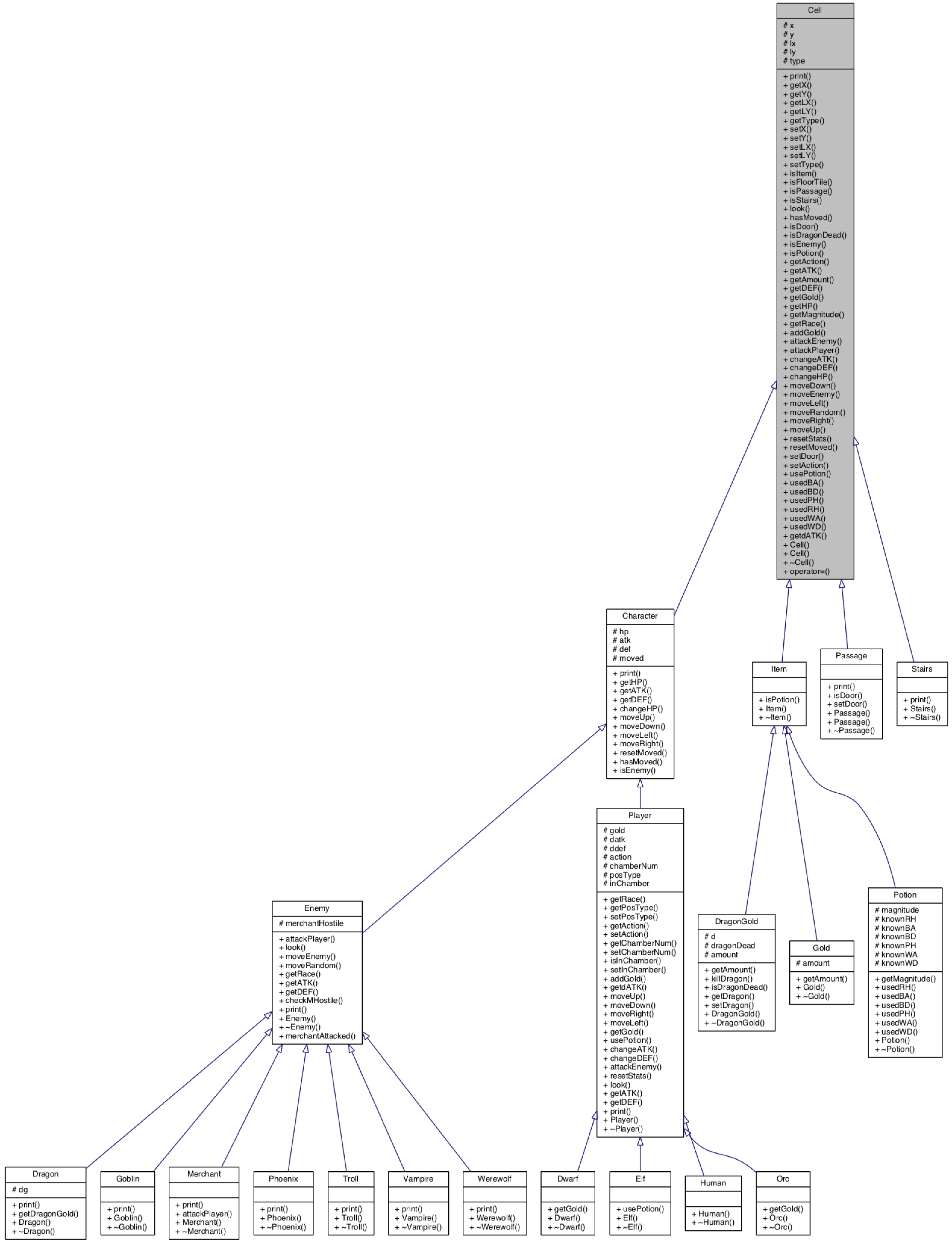
> What would you have done differently if you had the chance to start over?

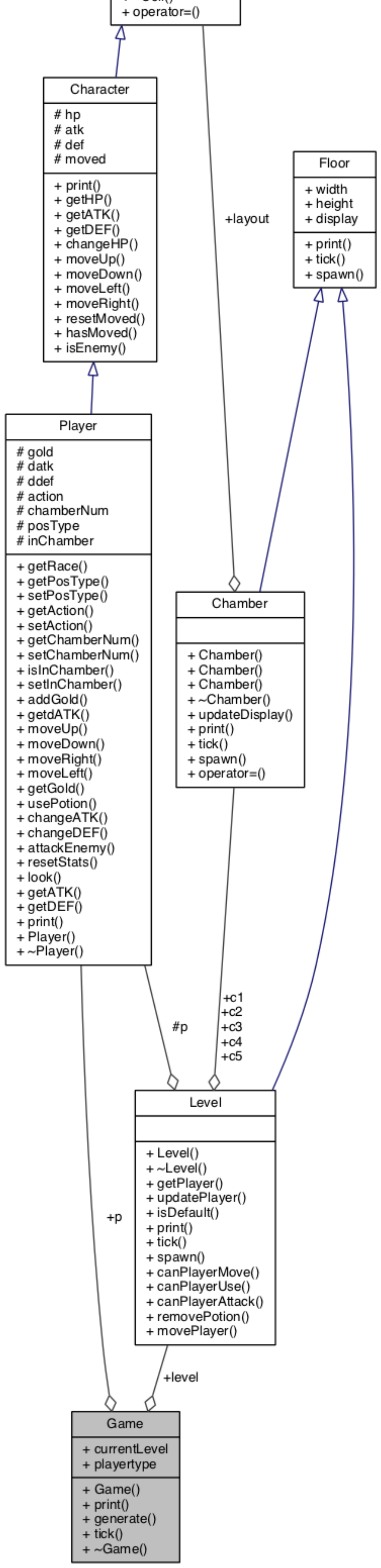
If we had the chance to start over, I think we both agree that we would've allocated a lot more time to handling the player's movement. We spent a good deal of time working on this mainly because of our implementation: passageways were separate from chambers, and movement in chambers was different outside of chambers.

Another thing we would have changed was allocating time to master dynamic casting. Dynamic casting caused a few bugs as we didn't do it correctly, so we decided to not include it at all.
However, all in all, we think this was a very good learning experience and we wouldn't start over even if we could, because we both learned a lot from this project.

4 UML

Our UML was pretty big, so I had to split it into two pages (on the next two)
Note: on the second page, the cut off box is the Cell that's on the first page.


Cell
# x
# y
# lx
# ly
# type
+ print()
+ getX()
+ getY()
+ getLX()
+ getLY()
+ getType()
+ setX()
+ setY()
+ setLX()
+ setLY()
+ setType()
+ isItem()
+ isFloorTile()
+ isPassage()
+ isStairs()
+ look()
+ hasMoved()
+ isDoor()
+ isDragonDead()
+ isEnemy()
+ isPotion()
+ getAction()
+ getATK()
+ getAmount()
+ getDEF()
+ getGold()
+ getHP()
+ getMagnitude()
+ getRace()
+ addGold()
+ attackEnemy()
+ attackPlayer()
+ changeATK()
+ changeDEF()
+ changeHP()
+ moveDown()
+ moveEnemy()
+ moveLeft()
+ moveRandom()
+ moveRight()
+ moveUp()
+ resetStats()
+ resetMoved()
+ setDoor()
+ setAction()
+ usePotion()
+ usedBA()
+ usedBD()
+ usedPH()
+ usedRH()
+ usedWA()
+ usedWD()
+ getdATK()
+ Cell()
+ Cell()
+ ~Cell()
+ operator=()
Character
# hp
# atk
# def
# moved
+ print()
+ getHP()
+ getATK()
+ getDEF()
+ changeHP()
+ moveUp()
+ moveDown()
+ moveLeft()
+ moveRight()
+ resetMoved()
+ hasMoved()
+ isEnemy()
Item
+ isPotion()
+ Item()
+ ~Item()
Passage
+ print()
+ isDoor()
+ setDoor()
+ Passage()
+ Passage()
+ ~Passage()
Stairs
+ print()
+ Stairs()
+ ~Stairs()
Player
# gold
# datk
# ddef
# action
# chamberNum
# posType
# inChamber
+ getRace()
+ getPosType()
+ setPosType()
+ getAction()
+ setAction()
+ getChamberNum()
+ setChamberNum()
+ isInChamber()
+ setInChamber()
+ addGold()
+ getdATK()
+ moveUp()
+ moveDown()
+ moveRight()
+ moveLeft()
+ getGold()
+ usePotion()
+ changeATK()
+ changeDEF()
+ attackEnemy()
+ resetStats()
+ look()
+ getATK()
+ getDEF()
+ print()
+ Player()
+ ~Player()
Enemy
# merchantHostile
+ attackPlayer()
+ look()
+ moveEnemy()
+ moveRandom()
+ getRace()
+ getATK()
+ getDEF()
+ checkMHostile()
+ print()
+ Enemy()
+ ~Enemy()
+ merchantAttacked()
DragonGold
# d
# dragonDead
# amount
+ getAmount()
+ killDragon()
+ isDragonDead()
+ getDragon()
+ setDragon()
+ DragonGold()
+ ~DragonGold()
Gold
# amount
+ getAmount()
+ Gold()
+ ~Gold()
Potion
# magnitude
# knownRH
# knownBA
# knownBD
# knownPH
# knownWA
# knownWD
+ getMagnitude()
+ usedRH()
+ usedBA()
+ usedBD()
+ usedPH()
+ usedWA()
+ usedWD()
+ Potion()
+ ~Potion()
Dragon
# dg
+ print()
+ getDragonGold()
+ Dragon()
+ ~Dragon()
Goblin
+ print()
+ Goblin()
+ ~Goblin()
Merchant
+ print()
+ attackPlayer()
+ Merchant()
+ ~Merchant()
Phoenix
+ print()
+ Phoenix()
+ ~Phoenix()
Troll
+ print()
+ Troll()
+ ~Troll()
Vampire
+ print()
+ Vampire()
+ ~Vampire()
Werewolf
+ print()
+ Werewolf()
+ ~Werewolf()
Dwarf
+ getGold()
+ Dwarf()
+ ~Dwarf()
Elf
+ usePotion()
+ Elf()
+ ~Elf()
Human
+ Human()
+ ~Human()
Orc
+ getGold()
+ Orc()
+ ~Orc()


+ Cell()
+ operator=()
Character
# hp
# atk
# def
# moved
+ print()
+ getHP()
+ getATK()
+ getDEF()
+ changeHP()
+ moveUp()
+ moveDown()
+ moveLeft()
+ moveRight()
+ resetMoved()
+ hasMoved()
+ isEnemy()
+layout
Floor
+ width
+ height
+ display
+ print()
+ tick()
+ spawn()
Player
# gold
# datk
# ddef
# action
# chamberNum
# posType
# inChamber
+ getRace()
+ getPosType()
+ setPosType()
+ getAction()
+ setAction()
+ getChamberNum()
+ setChamberNum()
+ isInChamber()
+ setInChamber()
+ addGold()
+ getdATK()
+ moveUp()
+ moveDown()
+ moveRight()
+ moveLeft()
+ getGold()
+ usePotion()
+ changeATK()
+ changeDEF()
+ attackEnemy()
+ resetStats()
+ look()
+ getATK()
+ getDEF()
+ print()
+ Player()
+ ~Player()
Chamber
+ Chamber()
+ Chamber()
+ Chamber()
+ ~Chamber()
+ updateDisplay()
+ print()
+ tick()
+ spawn()
+ operator=()
#p
+c1
+c2
+c3
+c4
+c5
Level
+ Level()
+ ~Level()
+ getPlayer()
+ updatePlayer()
+ isDefault()
+ print()
+ tick()
+ spawn()
+ canPlayerMove()
+ canPlayerUse()
+ canPlayerAttack()
+ removePotion()
+ movePlayer()
+p
+level
Game
+ currentLevel
+ playertype
+ Game()
+ print()
+ generate()
+ tick()
+ ~Game()