Partners: Jason Park (c39park) & Anusan Sivakumaran (a26sivak)
CS246 Assignment 5
CC3k
Design Document

Design Document

Main Classes

The game system uses an abstract superclass called Entity, and each player/enemy/item is implemented as a concrete subclass of the Entity class. The Player class extends the Entity class and each player race extends the Player class. This allows for unique player fields such as wallet (gold collected). Similarly, the Enemy class is the equivalent of Player for enemy races. Gold and Potions extend directly from the Entity class, each with their own unique fields such as value for gold and stat bonuses for potions. Gold also has a pointer to Dragon (and vice versa) to allow pickup of dragon hoard only if dragon is dead.

This scheme allows for adding additional classes and races with relative ease, as these classes simply need to be created and included assuming that they follow the structure of the Player/Enemy superclass. In other words, the implementation of these classes would require a simple inheritance followed by the implementation of unique fields and methods. This strategy also makes it easy to implement more features to all races.

Map/Grid

The map and its relevant functions were organized into the Map class. This class handled the map of the dungeon floor in a 2D array of pointers to Cells. The Cell class held fields indicating the type of map tile (wall, floor, passage, etc) as well as what is occupying it. The objects mentioned above were mostly accessed through the occupant pointers in these Cells.

Generation/Spawning

As all rooms have an equal chance of spawning anything, each floor tile (the only spawnable tile) of the map was organized into one of the 5 rooms. This information was held in a 2D array, with the first index being the room number and the second being the tile number of that room. Thus when each Entity was spawned, each room had an equal probability of spawning it, and each tile in each room had an equal probability as well.

The player was spawned by picking a random room and random tile. The stairs were spawned by picking a random room that the player did not spawn in, and a random tile. Items (both gold and potions) were spawned in random rooms with random tiles (10 each) with the specified probabilities given. For each dragon hoard, a dragon was also spawned and a counter was updated so that during enemy spawn, exactly 20 enemies are spawned total. Enemies were spawned similarly to items, with each enemy spawning at the correct probability.

Each spawn checks the Cell to ensure it is not already occupied before spawning, and the probabilities are handled by distributing a specific portion of a

random variable's range to each potential spawn (eg. for a 25% chance, 1 number out of 4 possible numbers denotes the spawning of something).

Design Patterns
The visitor design pattern was used to implement the combat interactions. The player and enemy classes have pure virtual methods called hit() and getHit(). Every concrete class has unique implementations of these classes that determine various combat scenarios (vampires regenerate hp, orcs do 50% more damage to goblins). General polymorphism and inheritance concepts were also used to effectively organize the various methods of the different races, such as common HP, Atk, Def getters and setters for all characters.

The original plan considered using the singleton design pattern to ensure only one player existed during the game. However it was decided that it was not necessary as the Player constructor is only ever called in the starting menu of selecting the race, so practically no risk of accidentally creating a second Player.

Another design pattern that was also considered was the decorator pattern for use of potions by the player. During the game the player will encounter items to pick up and originally it was thought that the decorator plan would make it much easier to deal with this scenario. However this design pattern was discarded because only the player was picking up potions, which is only one feature that enhances the player class in a very simple way (temporary stat changes). Thus the solution was to simply create a method in potion class that would be called once a player picks up a potion. This method would alter stat variables separate from the player's base stats. These temporary stat variables are cleared at the end of each floor so that their stats return to base values.

Display
A basic display class was made to get information about player health and stats as well as floor number and gold. This information along with the current state of the map are printed with the print method in this class. The map is printed by having any Cell occupants have their corresponding symbols printed instead of the Cell they are on (eg. "E" for elf is printed instead of a "." for floor)

Death and Enemy AI
Status is a class that checks whether each character is still living, as well as controls the movement of enemies through a method called update which runs every turn. If the HP of any enemy drops below 0, it is correctly removed from the map and any resulting methods are called (dead dragon releases its hoard, enemies drop correct amount of gold). A dead player ends the game and gives the user a chance to restart.

Update also goes through the entire map looking for enemies that will move or attack (attack if a player is in range). Merchants attack based on a boolean that

determines merchant hostility (which updates if a merchant is attacked) and dragons never move. Dragons also attack if update recognizes that a player is next to a dragon hoard.

Enemy movement methods are organized into a simple static class called Mover.

Endgame

The game ends when the player dies, the user quits/restarts, or the player reaches the stairs 5 times (implying the end of the fifth floor). Each floor is simply implemented with a for loop that runs 5 times, and loops every time the player reaches the location of the stairs.

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?

Over the course of developing CC3k, there were many lessons my partner and I learned about developing software in teams. The first lesson being that time management is key. By organizing this project into specific parts with a designated time interval allowed us to maximize our time and meet the requirements.

Another important lesson learned through this assignment was teamwork and collaboration. Working as a team we learned to coordinate and collaborate our ideas into this game. Not only was it just an assignment, but also more of a learning experience as to applying the techniques learned in lectures to real life applications.

The final lesson we learned is to always have a backup plan when all else fails. While developing certain features of CC3k we learned that some our proposed solutions became far too complicated or ended up not working. This lead to spending more time trying to fix the problem or search for an alternate solution. By having an alternate solution prepared ahead of time, it would maximize our efficiency.

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

One thing we could have done differently given the chance to start over is to spend more time planning. Initially, we felt that we had planned out everything, however we did come to specific areas of the game that we didn't consider. For example in our original UML we had declared cell being an abstract class, and there would subclasses for all the different types of cells. This was unnecessary because you could simply store this information through private fields (string

variable). There were several other instances similar to this problem, which lead to a lot of time being spent on redesigning and implementing these classes.

Another crucial thing we could have done differently is to not overthink while we are planning. At some points in the development the ideas we had proposed become unrealistic or far too complex to implement. Thus more time had to be spent redesigning and implementing.

Questions & Answers (from due date 1)

How could your design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?

The system could have a superclass called *Player* that is abstract, and each of the different races could be concrete subclasses of the *Player* class. The *Player* class would have the common fields and methods that all PC's need, such as HP, Atk, Def, move(string dir), and pure virtual methods that depend on the race such as attack(). Each individual race class would then have constants that determine its starting HP, Atk, Def. They would also have implementations of the

previous pure virtual methods that would give the class all of its unique features (eg. using the visitor design pattern to implement the unique combat interactions between different races).

This system allows adding additional classes/races with relative ease as the new race classes simply need to be included assuming they follow the structure of the other *Player* races. This would mean that in the future an additional class would require a simple inheritance (assuming that the race has the same attributes as the abstract Player class) followed by implementations of unique methods.

How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?
The generation of enemies is very similar to the generation of the player character, however there are some key differences.

First of all, the *Player* object should be implemented with the singleton design pattern as there should never be more than 1 PC in the game at any time, while there can be many *Enemy* objects at the same time.

Secondly, there are certain fields and methods that must be implemented differently for the enemies. For example as enemies move randomly (if at all) their move method does not need an argument for direction.

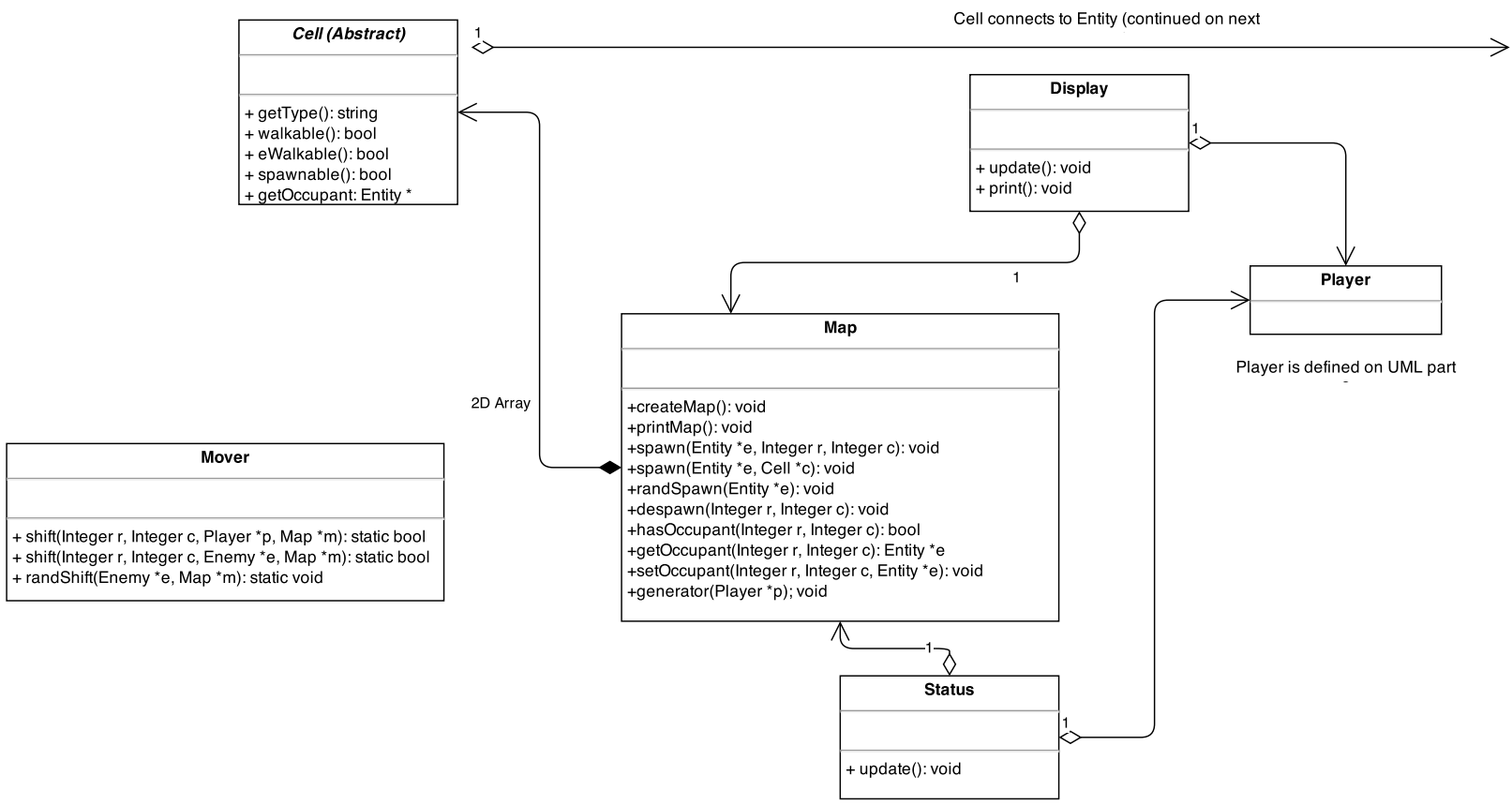
How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain. The abilities of the enemies are primarily interaction based (ie related to combat) so they could be implemented by using the visitor design pattern to create unique attack() methods for each interaction. For example, when a vampire is attacking a dwarf, it needs to lose 5 HP rather than gain).

PC's also have interaction based abilities, however, they also have abilities that are related to potion use (drow) or that take effect in every turn regardless of combat (troll). These would need unique implementations, such as adding a decorator to change the effects of potions or making additional methods that are called at the end of every turn.

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?
The decorator design pattern would be beneficial in implementing the temporary effects of potions as it would allow an easy way of manipulating the Atk and Def values during combat without changing the base values. These decorators can also easily be deleted at the end of a floor thus returning the PC's Atk and Def back to its base values.

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?

Gold and Potions can be subclasses of an abstract superclass called *Item*. The implementation for the generation/spawn of Gold and Potions are identical other than their respective probabilities and total quantity on a given floor. Thus, the methods for generation can share an implementation in the *Item* class with only the probability and quantity differences implemented in their individual classes.



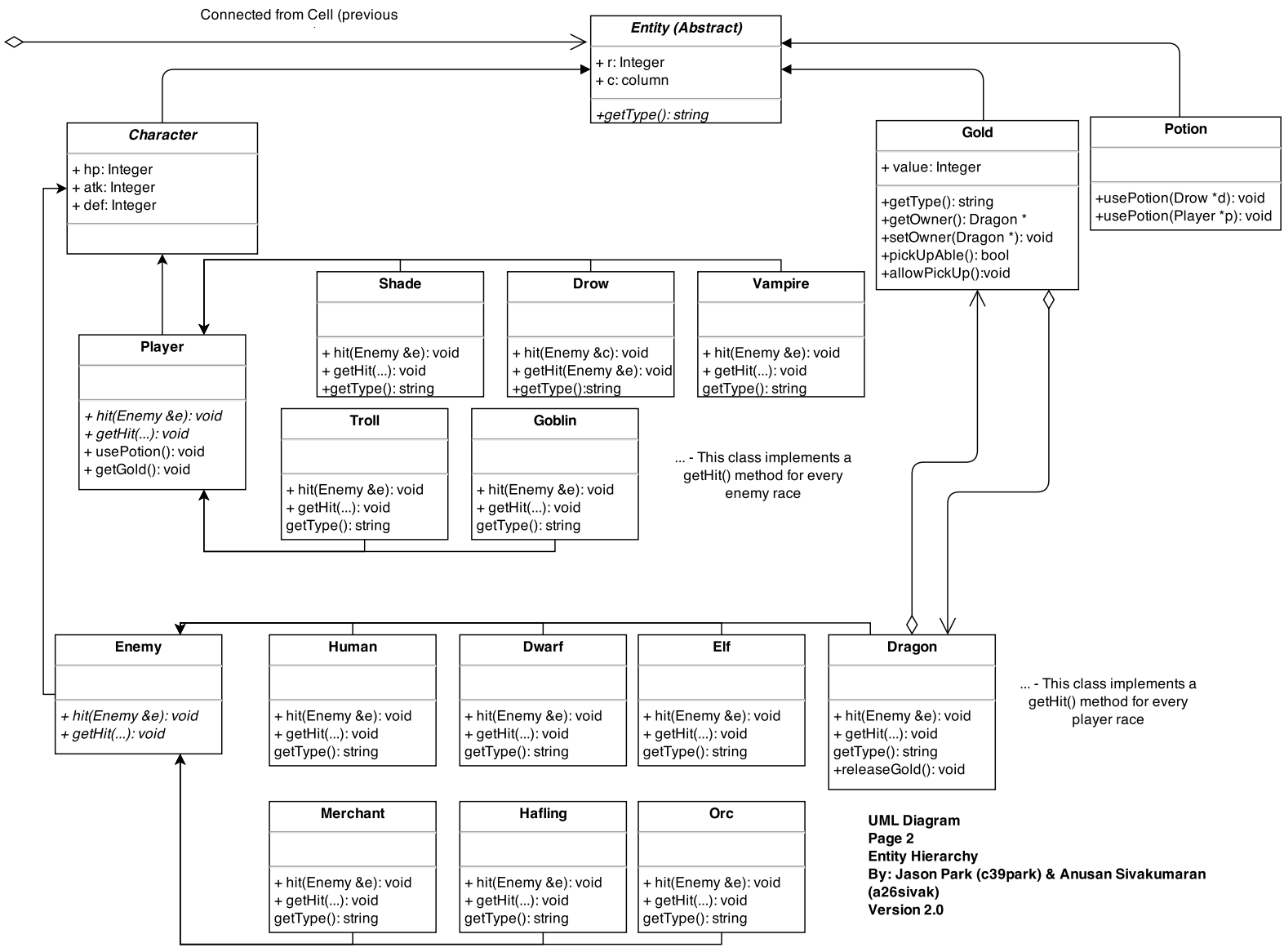
UML Diagram

Page 1

Cell Hierarchy

By: Jason Park (c39park) & Anusan Slvakumaran (a26sivak)

Version 2.0


Connected from Cell (previous
Entity (Abstract)
+ r: Integer
+ c: column
+getType(): string
Character
+ hp: Integer
+ atk: Integer
+ def: Integer
Gold
+ value: Integer
+getType(): string
+getOwner(): Dragon *
+setOwner(Dragon *): void
+pickUpAble(): bool
+allowPickUp():void
Potion
+usePotion(Drow *d): void
+usePotion(Player *p): void
Shade
+ hit(Enemy &e): void
+ getHit(...): void
+getType(): string
Drow
+ hit(Enemy &c): void
+ getHit(Enemy &e): void
+getType():string
Vampire
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Player
+ hit(Enemy &e): void
+ getHit(...): void
+ usePotion(): void
+ getGold(): void
Troll
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Goblin
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
... - This class implements a getHit() method for every enemy race
Enemy
+ hit(Enemy &e): void
+ getHit(...): void
Human
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Dwarf
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Elf
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Dragon
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
+releaseGold(): void
... - This class implements a getHit() method for every player race
Merchant
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Hafling
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
Orc
+ hit(Enemy &e): void
+ getHit(...): void
getType(): string
UML Diagram
Page 2
Entity Hierarchy
By: Jason Park (c39park) & Anusan Sivakumaran (a26sivak)
Version 2.0