Design Document – sdsuo/l2ke

Project Design

Overview

Our implementation of cc3k uses a Game singleton which controls the general direction of the game (e.g. starting the game, choosing player type, restarting, quitting, etc.). It does this by managing all the keyboard input and calling different methods based on them. The idea is that the Game can delegate which events should occur, and a chain of calls through various classes can get closer and closer to the specific event. For example, if the player enters `no`, Game will call `player->move([Cell north of player])`. Then in `Player::move`, it will check whether a player can step on that cell, and if so, will move to it via a call to `Character::move`. This way, specific/special cases that involve various factors (e.g. Orc doing 1.5x damage only on Goblin) are resolved by smaller classes, making the code very modular. We will go into specifics in the following sections.

Spawning

Random Tile

The entire random spawning process is done by Floor. We begin with choosing a random Tile to spawn on. Floor chooses a random Chamber and then Chamber chooses a random, unoccupied Tile (out of the tiles in that specific chamber) by returning a pointer to that Tile. This process is the exact same for spawning any type of entity. Floor can then spawn each type of Entity any number of times, in whatever order we need.

Random Entity

In terms of determining what to spawn, the Tile class has methods to spawn each type of Entity. For Stair and Player, we use the Singleton pattern (since there is only one of each) to return the respective instance. Otherwise, we are using the Factory method pattern to generate the Entity. The factory methods encapsulate the probabilities of spawning different subclasses. This design is slightly different than our original plan as we moved the spawning of specific types of Entities from the Cell class to the Tile class. This is because Tiles are the only Cell type that should have things spawning on them.

Movement

Movement Restriction

Because Player can move on certain Cells while Enemy cannot, we use a general method in Cell that takes in a character, and to determine whether that character can move on it. This method is overwritten by the various subclasses of Cells so that the various ways of stepping (Player onto Doorway, Enemy onto passage, etc.) are determined by the specific Cell types. This keeps the logic for movement restriction modular. This means that if we added a new Cell type, we would just need to overwrite this method for taking in Player and Enemy.

Actual Movement

Since the actual moving of a Player and Enemy are the same, we use a `move(Cell *)` method in Character (superclass of the Player and Enemy) to deal with clearing the current Cell and placing the Player/Enemy on the new Cell. This is done through the Observer pattern. The Entity notifies the Cell it's on that it has moved, so the Cell can clear its Entity pointer.

Potions

u <direction>

When a Player attempts to "use" in a direction, the Cell is notified through the `Cell::pickedUp(Player *)` method. This method checks whether it has an Entity, and if it does, it calls the virtual `Entity::pickup` method. This method itself does nothing. Doing this, we can filter out any "misuses" (using on any Cell without a Potion). If it is a Potion, however, the `Potion::pickup` method will be called instead. Thus, we are using inheritance and virtual methods to delegate the outcome of "use".

Tracking Potions

We are using the Decorator Design Pattern to track the effects of the Potions. Thus, Potion is a subclass of Player. So, when a Player picks up a Potion, the Player itself is wrapped in the Potion and the Potion becomes the current player. When we wrap the Player, the effect of the different types of potions is implemented. Once the floor has been cleared, we unwrap the player to get back the original Player. In this unwrapping process, each of the Potions will undo their effects (if needed).

Combat

Overview

Similar to the `Cell::pickedUp` method for potions, we use the `Cell::isAttackedBy` method to determine whether the Entity the Cell holds (if any) can be attacked. If it can, then `Entity::isAttackedBy` is called. To model the different pairings of races in combat, we use the Visitor Design Pattern. Character has different overloaded virtual methods of `isAttacked` and `attack`, which take in a certain subclasses of either Player or Enemy. These methods are overwritten by these subclasses, which will react differently based on the attacker or attacked. Character also has generic `isAttacked` and `attack` methods, which take in a Character * and do default actions. For example, if an Orc attacks a Goblin, `Orc::attack(Goblin*)` (overwritten from `Character::attack(Goblin *)`) is called, which will calculate the damage 1.5x more as it is a Goblin. It this Orc were to attack a Drow instead, then the generic `Character::attack(Character *)` will be called (since `Character::attack(Drow *)` doesn't exist). Thus, various race pairings are modelled differently in each of the subclasses themselves.

Changes

To keep our code even more modular, and to continue with the idea of the Visitor Design Pattern, we added `Character::calculateDamageOn`, `Character::calculateGoldFrom`, and `Character::killedBy` methods to produce different outcomes based on the Character type passed in. These work just like

`Character::isAttacked` and `Character::attack`, with subclasses overriding this method if needed. So for the above example with the Orc, after Orc::attack is called, `Orc::calculateDamageOn(Goblin *)` will be called to give the 1.5x damage value needed.

Gold

The `isSteppedOnBy` method in Cell is called when a Player moves onto it. By default this method does nothing. Different Treasure types then overwrite this method to add gold to the Player.

Design Questions Revisited:

1. **[Question]** 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?

   [Answer] We used inheritance to organize our different races. Our superclass Character had fields common to all races. Extending off this class we had Enemy and Player, which each had children which were their different types. This makes generating classes easy, since the constructor in Character can deal with initializing many common attributes. Adding additional classes would be an easy change. We would simply inherit off either Player or Enemy.

2. **[Question]** How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

   [Answer] Our system used the Factory Design Pattern to generate enemies, as there can be multiple instances of them. We had a factory method that encapsulated the different probabilities for each enemy, and generated an enemy based on that. On the other hand, the player character was generated with the Singleton Design Pattern, as there is only one instance of the player object. The race of the player was already set beforehand, so that when the getInstance of player was called, the specific race of Player would be returned. The way we generate enemies and the player is similar in the sense that there is one method delegating which race to be returned. However, the generation of player is different as the same player is always returned whilst for the enemy, a new random enemy is returned every call.

3. **[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 the Visitor Design Pattern to implement the abilities for the enemy races. Based on the type of Enemy being attacked or the type of Player attacking, the specific enemy race will respond in different ways based on their abilities. Thus the enemy character's abilities are captured in the specific subclasses themselves. This is the same

technique we used for the player character, except that it is extended to other events based around the character. This includes picking up a potion, invoking a passive ability (i.e. +5 HP a turn for a Troll), etc. These are used to capture abilities not directly related to combat, which is something enemies do not have.

4. **[Question]** 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?

   [Answer] We used the Decorator Design pattern so that the Potion itself will wrap around the Player. Thus, multiple effects can be stacked and implicitly tracked. To remove the temporary status effects, we retrieved the underlying player object by recursively unwrapping the decorators.

5. **[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] The floor generated specific items by selecting a random chamber and a random cell. It then delegated the task of item generation to the individual tiles. The tiles generate specific items with a Factory Design Pattern. Code will be reused, as finding the random cell to spawn an item will be the same for any item.

Reflection:

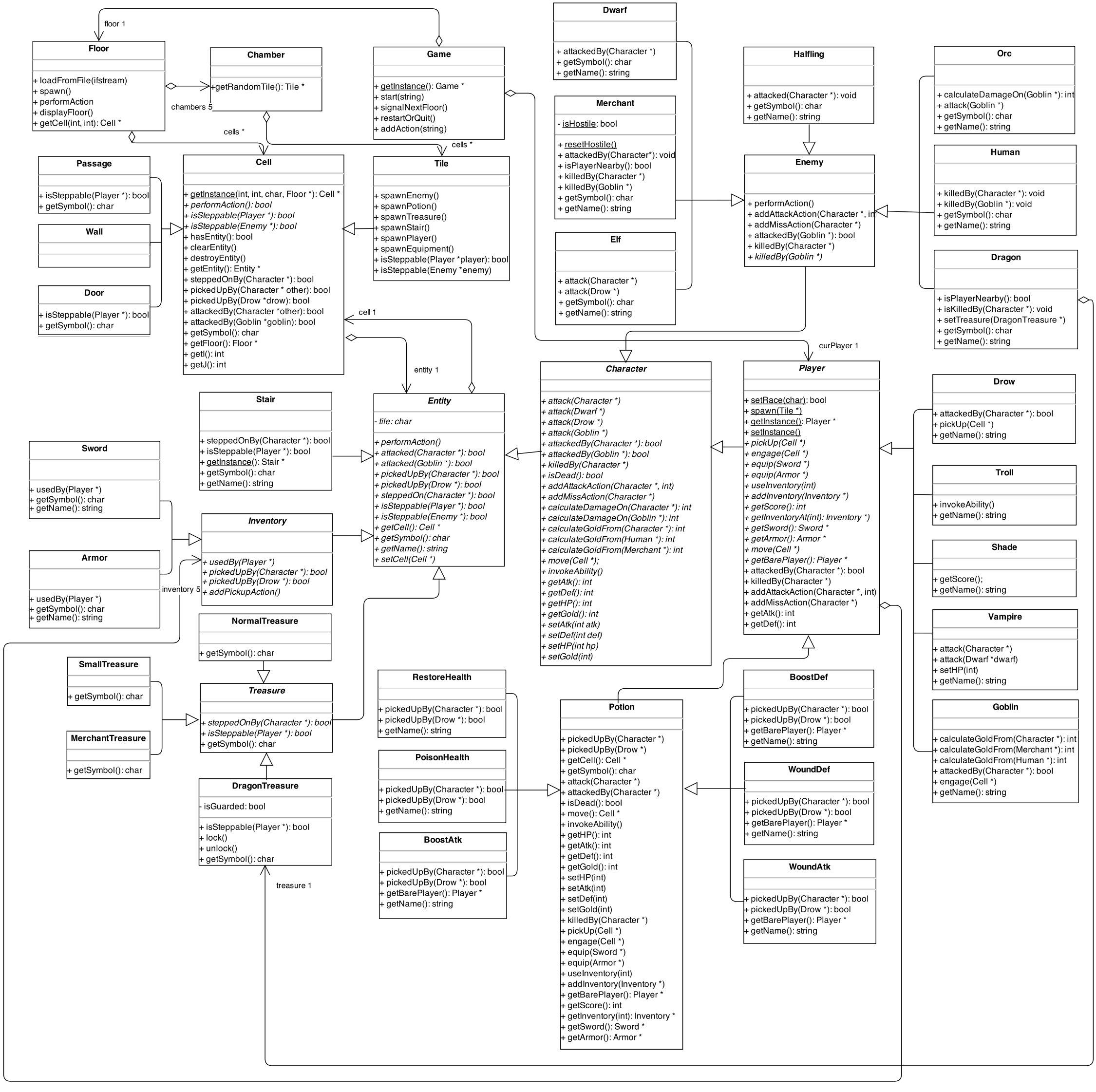
1. **[Question]** What lessons did this project teach you about developing software in teams?

   [Answer] One thing we learned was the importance of team member communication. In the beginning, we weren't as vocal with our code changes as we could have been. As the project passed, we learned to communicate changes more often and in more detail. This allowed each of us to get a good sense of how the project was coming along, making us more effective in further implementing code. We used git for our source code management, so another thing that we learned was to commit more often. This made changes more incremental and easier to follow. This also goes along with the first point of communicating changes more effectively.

2. **[Question]** What would you have done differently if you had the chance to start over?

   [Answer] We had a number of issues in terms of memory management as we were trying to wrap up our project. What we would have done differently to mitigate this would be to have started running our program with valgrind from the very beginning as well as more frequently. This way we could fix memory errors earlier on before they pile up. We also would have taken the time before starting to code to write tests for expected behaviour.

These wouldn’t be automated, but would simply list all the specific behaviours we needed to have. Then, when we were adding code, we would have gone through these tests and ensured that they were all working (if the feature had been added). This way, we could be certain that an additional change was not breaking another one that had been added previously. We also would have done more research on our DLC to get a better estimate of how long they would have taken, and planned accordingly.


floor 1
Floor
+ loadFromFile(ifstream)
+ spawn()
+ performAction
+ displayFloor()
+ getCell(int, int): Cell *
chambers 5
Chamber
+getRandomTile(): Tile *
cells *
Game
+ getInstance(): Game *
+ start(string)
+ signalNextFloor()
+ restartOrQuit()
+ addAction(string)
cells *
Dwarf
+ attackedBy(Character *)
+ getSymbol(): char
+ getName(): string
Merchant
- isHostile: bool
+ resetHostile()
+ attackedBy(Character*): void
+ isPlayerNearby(): bool
+ killedBy(Character *)
+ killedBy(Goblin *)
+ getSymbol(): char
+ getName(): string
Elf
+ attack(Character *)
+ attack(Drow *)
+ getSymbol(): char
+ getName(): string
Halfling
+ attacked(Character *): void
+ getSymbol(): char
+ getName(): string
Orc
+ calculateDamageOn(Goblin *): int
+ attack(Goblin *)
+ getSymbol(): char
+ getName(): string
Human
+ killedBy(Character *): void
+ killedBy(Goblin *): void
+ getSymbol(): char
+ getName(): string
Enemy
+ performAction()
+ addAttackAction(Character *, in
+ addMissAction(Character *)
+ attackedBy(Goblin *): bool
+ killedBy(Character *)
+ killedBy(Goblin *)
Dragon
+ isPlayerNearby(): bool
+ isKilledBy(Character *): void
+ setTreasure(DragonTreasure *)
+ getSymbol(): char
+ getName(): string
Passage
+ isSteppable(Player *): bool
+ getSymbol(): char
Wall
Door
+ isSteppable(Player *): bool
+ getSymbol(): char
Cell
+ getInstance(int, int, char, Floor *): Cell *
+ performAction(): bool
+ isSteppable(Player *): bool
+ isSteppable(Enemy *): bool
+ hasEntity(): bool
+ clearEntity()
+ destroyEntity()
+ getEntity(): Entity *
+ steppedOnBy(Character *): bool
+ pickedUpBy(Character * other): bool
+ pickedUpBy(Drow *drow): bool
+ attackedBy(Character *other): bool
+ attackedBy(Goblin *goblin): bool
+ getSymbol(): char
+ getFloor(): Floor *
+ getI(): int
+ getJ(): int
Tile
+ spawnEnemy()
+ spawnPotion()
+ spawnTreasure()
+ spawnStair()
+ spawnPlayer()
+ spawnEquipment()
+ isSteppable(Player *player): bool
+ isSteppable(Enemy *enemy)
cell 1
entity 1
curPlayer 1
Character
+ attack(Character *)
+ attack(Dwarf *)
+ attack(Drow *)
+ attack(Goblin *)
+ attackedBy(Character *): bool
+ attackedBy(Goblin *): bool
+ killedBy(Character *)
+ isDead(): bool
+ addAttackAction(Character *, int)
+ addMissAction(Character *)
+ calculateDamageOn(Character *): int
+ calculateDamageOn(Goblin *): int
+ calculateGoldFrom(Character *): int
+ calculateGoldFrom(Human *): int
+ calculateGoldFrom(Merchant *): int
+ move(Cell *);
+ invokeAbility()
+ getAtk(): int
+ getDef(): int
+ getHP(): int
+ getGold(): int
+ setAtk(int atk)
+ setDef(int def)
+ setHP(int hp)
+ setGold(int)
Player
+ setRace(char): bool
+ spawn(Tile *)
+ getInstance(): Player *
+ setInstance()
+ pickUp(Cell *)
+ engage(Cell *)
+ equip(Sword *)
+ equip(Armor *)
+ useInventory(int)
+ addInventory(Inventory *)
+ getScore(): int
+ getInventoryAt(int): Inventory *)
+ getSword(): Sword *
+ getArmor(): Armor *
+ move(Cell *)
+ getBarePlayer(): Player *
+ attackedBy(Character *): bool
+ killedBy(Character *)
+ addAttackAction(Character *, int)
+ addMissAction(Character *)
+ getAtk(): int
+ getDef(): int
Drow
+ attackedBy(Character *): bool
+ pickUp(Cell *)
+ getName(): string
Troll
+ invokeAbility()
+ getName(): string
Shade
+ getScore();
+ getName(): string
Vampire
+ attack(Character *)
+ attack(Dwarf *dwarf)
+ setHP(int)
+ getName(): string
Goblin
+ calculateGoldFrom(Character *): int
+ calculateGoldFrom(Merchant *): int
+ calculateGoldFrom(Human *): int
+ attackedBy(Character *): bool
+ engage(Cell *)
+ getName(): string
Stair
+ steppedOnBy(Character *): bool
+ isSteppable(Player *): bool
+ getInstance(): Stair *
+ getSymbol(): char
+ getName(): string
Entity
- tile: char
+ performAction()
+ attacked(Character *): bool
+ attacked(Goblin *): bool
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ steppedOn(Character *): bool
+ isSteppable(Player *): bool
+ isSteppable(Enemy *): bool
+ getCell(): Cell *
+ getSymbol(): char
+ getName(): string
+ setCell(Cell *)
Sword
+ usedBy(Player *)
+ getSymbol(): char
+ getName(): string
Inventory
+ usedBy(Player *)
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ addPickupAction()
Armor
+ usedBy(Player *)
+ getSymbol(): char
+ getName(): string
inventory 5
NormalTreasure
+ getSymbol(): char
SmallTreasure
+ getSymbol(): char
MerchantTreasure
+ getSymbol(): char
Treasure
+ steppedOnBy(Character *): bool
+ isSteppable(Player *): bool
+ getSymbol(): char
DragonTreasure
- isGuarded: bool
+ isSteppable(Player *): bool
+ lock()
+ unlock()
+ getSymbol(): char
treasure 1
RestoreHealth
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ getName(): string
PoisonHealth
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ getName(): string
BoostAtk
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ getBarePlayer(): Player *
+ getName(): string
Potion
+ pickedUpBy(Character *)
+ pickedUpBy(Drow *)
+ getCell(): Cell *
+ getSymbol(): char
+ attack(Character *)
+ attackedBy(Character *)
+ isDead(): bool
+ move(): Cell *
+ invokeAbility()
+ getHP(): int
+ getAtk(): int
+ getDef(): int
+ getGold(): int
+ setHP(int)
+ setAtk(int)
+ setDef(int)
+ setGold(int)
+ killedBy(Character *)
+ pickUp(Cell *)
+ engage(Cell *)
+ equip(Sword *)
+ equip(Armor *)
+ useInventory(int)
+ addInventory(Inventory *)
+ getBarePlayer(): Player *
+ getScore(): int
+ getInventory(int): Inventory *
+ getSword(): Sword *
+ getArmor(): Armor *
BoostDef
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ getBarePlayer(): Player *
+ getName(): string
WoundDef
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ getBarePlayer(): Player *
+ getName(): string
WoundAtk
+ pickedUpBy(Character *): bool
+ pickedUpBy(Drow *): bool
+ getBarePlayer(): Player *
+ getName(): string