Overview

Main Loop

- The main loop handles I/O, and passes in an istream and ostream to game
- The main loop takes care of game creation / restarting, and lets the user know of their game status (game over / victory)
- The main loop reads in the template maps and passes it onto the game
- The main loop **owens *Game***, and deletes it at the end

Game

- The Game class handles logic that persists throughout the game
- Game **owens *Board*** and ***Player***
- Game handles map generation
    - Game takes in an input stream from main, and generates a new ***Board***
- Game keeps track of the status of the merchants (if they are hostile or not)
- Game keeps track of the current level, and the status of the game (if player has won or not)
- Game handles player interactions
    - Player movement
    - Player attack
    - Player item interaction

Board

- The Board class owns the game map consisting of a 2D array of ***Tile***
- Board keeps track of the location of ***Player*** and Stairs
- Board keeps track of tiles that make up the Chambers for ***Game*** to generate entities
- Board has a frame count that makes sure that all entities are updated synchronously

Tile

- The tile class is owned by ***Board***
- Tiles make up the map — hence, tile describes each “tile” on the map, containing information of what entities are on said tile
- Tile consists of:
    - Location - location of the tile on the map
    - mapTile - the base character of the tile (floor, wall, etc.)
    - Treasure - Retrievable or nullptr if there are no treasures on the tile
    - Item - Interactable or nullptr if there are no items on the tile
    - Enemy - Enemy or nullptr if there are no enemies on the tile

    - Player - Player or nullptr if Player is not on the tile
    - ChamberId - ID of the chamber that the tile is part of (0 if it's not part of a chamber)

Character

- The character class is the abstraction that is used to construct the playable characters and enemies
- The character class takes care of interactions between entities
    - These are overridable, as some entities have different mechanisms
- Enemy
    - The enemy class constructs all enemies that are not Merchants and Dragons
    - Behavior
        - Enemies attack if a player is in its radius
        - Enemies walk / move one tile if a player is not in its radius
    - All enemies have an age to ensure that updates are synchronized and happens once per frame for all entities
    - Dragon
        - Dragons inherit from ***Enemy***
        - Dragons are not able to walk
        - Dragons are associated with a ***Retrievable*** which it protects
        - Dragons allow Retrievables they protect to be retrieved once they are defeated
    - Merchant
        - Merchants inherit from Enemy
        - Merchants are not aggressive by default, so they choose to Enemy::Walk when they are not aggressive
        - Merchants are created with a Merchant Hoard
- Player
    - The player class is an abstraction for the playable characters
    - Player has it's own beAttacked method, so that it can take less damage when player picks up the ***BarrierSuit***
    - Player methods getScore, pickupGold and usePotion which are overridable
    - Logs are owned by player, and are displayed after the map renders
    - Tracks a log of actions that happen, to print after every render. This includes when they see potions, or move in directions. If they are attacked, enemies will add to the log themselves.

Retrievable

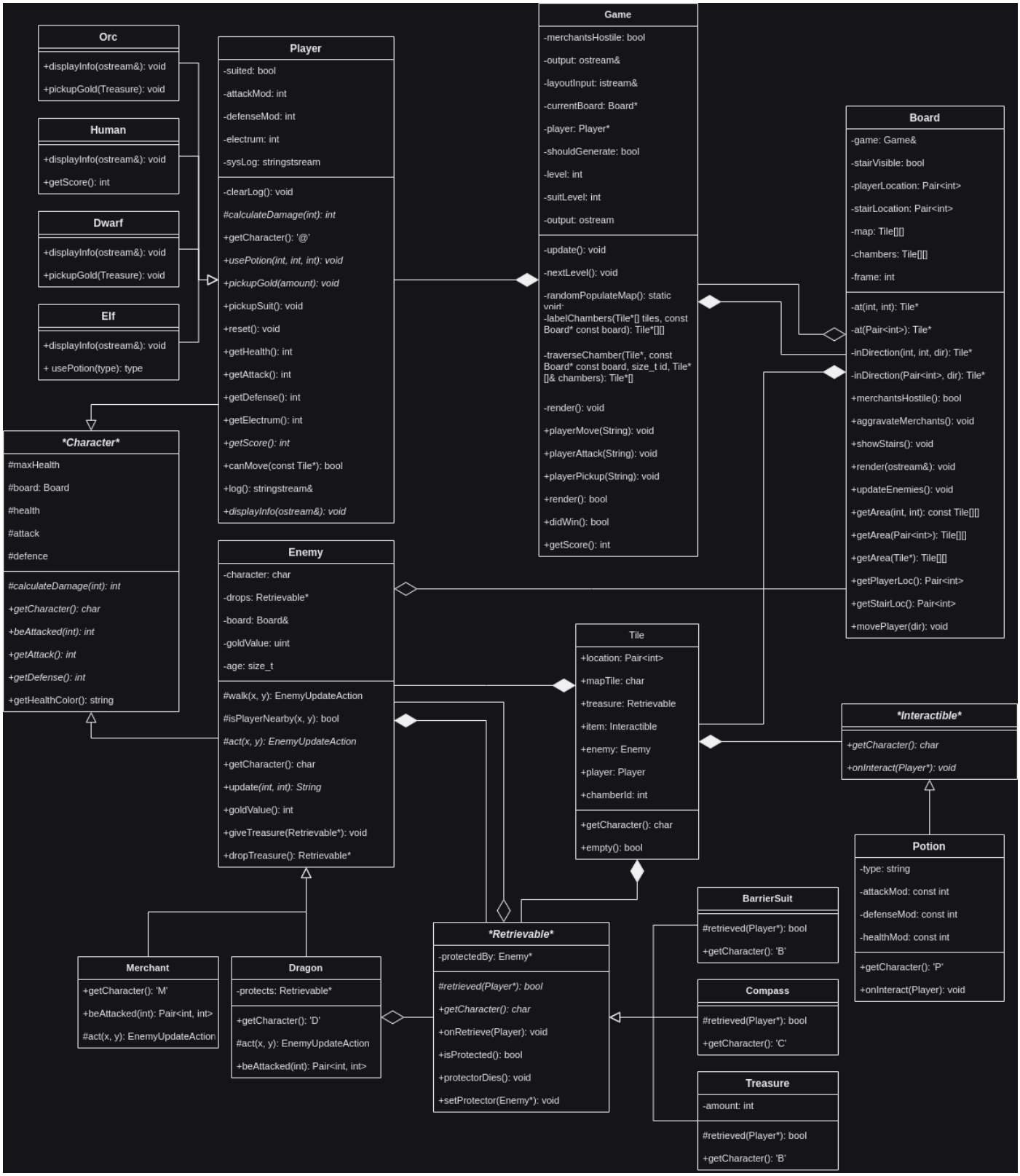
- Retrievable is an object class for things which are "retrieved" when walked upon

- Sometimes they are protected by an entity (such as a dragon), and so the onRetrieve returns a bool for whether the retrieve was successful. If it's still protected, it cannot be retrieved.
- We abstract out what the item does to the player on retrieving by passing in a pointer to the player, so the retrievable decides what happens. Such as gaining gold or suiting up
- If the onRetrieve returns true, the game will know to delete it and allow the player to move there. Otherwise the player is not allowed to move there.

Interactable

- Interactable is a class of items that need to be interacted with from a distance. Currently the only such class is a potion.
- When a potion is interacted with, like the retrievable, we pass a pointer to the player. The potion will then tell the player what its effects are, and add an entry to the log including the potion's identifier.
- Currently potions are all constructed by passing in all their relevant fields. No subclasses are used due to the similarity of how they act.

UML


Orc
+displayInfo(ostream&): void
+pickupGold(Treasure): void
Human
+displayInfo(ostream&): void
+getScore(): int
Dwarf
+displayInfo(ostream&): void
+pickupGold(Treasure): void
Elf
+displayInfo(ostream&): void
+ usePotion(type): type
Player
-suited: bool
-attackMod: int
-defenseMod: int
-electrum: int
-sysLog: stringstsream
-clearLog(): void
#calculateDamage(int): int
+getCharacter(): '@'
+usePotion(int, int, int): void
+pickupGold(amount): void
+pickupSuit(): void
+reset(): void
+getHealth(): int
+getAttack(): int
+getDefense(): int
+getElectrum(): int
+getScore(): int
+canMove(const Tile*): bool
+log(): stringstream&
+displayInfo(ostream&): void
Game
-merchantsHostile: bool
-output: ostream&
-layoutInput: istream&
-currentBoard: Board*
-player: Player*
-shouldGenerate: bool
-level: int
-suitLevel: int
-output: ostream
-update(): void
-nextLevel(): void
-randomPopulateMap(): static void
-labelChambers(Tile*[] tiles, const Board* const board): Tile*[][]
-traverseChamber(Tile*, const Board* const board, size_t id, Tile*[]& chambers): Tile*[]
-render(): void
+playerMove(String): void
+playerAttack(String): void
+playerPickup(String): void
+render(): bool
+didWin(): bool
+getScore(): int
Board
-game: Game&
-stairVisible: bool
-playerLocation: Pair<int>
-stairLocation: Pair<int>
-map: Tile[][]
-chambers: Tile[][]
-frame: int
-at(int, int): Tile*
-at(Pair<int>): Tile*
-inDirection(int, int, dir): Tile*
-inDirection(Pair<int>, dir): Tile*
+merchantsHostile(): bool
+aggravateMerchants(): void
+showStairs(): void
+render(ostream&): void
+updateEnemies(): void
+getArea(int, int): const Tile[][]
+getArea(Pair<int>): Tile[][]
+getArea(Tile*): Tile[][]
+getPlayerLoc(): Pair<int>
+getStairLoc(): Pair<int>
+movePlayer(dir): void
Character
#maxHealth
#board: Board
#health
#attack
#defence
#calculateDamage(int): int
+getCharacter(): char
+beAttacked(int): int
+getAttack(): int
+getDefense(): int
+getHealthColor(): string
Enemy
-character: char
-drops: Retrievable*
-board: Board&
-goldValue: uint
-age: size_t
#walk(x, y): EnemyUpdateAction
#isPlayerNearby(x, y): bool
#act(x, y): EnemyUpdateAction
+getCharacter(): char
+update(int, int): String
+goldValue(): int
+giveTreasure(Retrievable*): void
+dropTreasure(): Retrievable*
Tile
+location: Pair<int>
+mapTile: char
+treasure: Retrievable
+item: Interactible
+enemy: Enemy
+player: Player
+chamberId: int
+getCharacter(): char
+empty(): bool
Interactible
+getCharacter(): char
+onInteract(Player*): void
Potion
-type: string
-attackMod: const int
-defenseMod: const int
-healthMod: const int
+getCharacter(): 'P'
+onInteract(Player): void
BarrierSuit
#retrieved(Player*): bool
+getCharacter(): 'B'
Compass
#retrieved(Player*): bool
+getCharacter(): 'C'
Treasure
-amount: int
#retrieved(Player*): bool
+getCharacter(): 'B'
Retrievable
-protectedBy: Enemy*
#retrieved(Player*): bool
+getCharacter(): char
+onRetrieve(Player): void
+isProtected(): bool
+protectorDies(): void
+setProtector(Enemy*): void
Merchant
+getCharacter(): 'M'
+beAttacked(int): Pair<int, int>
#act(x, y): EnemyUpdateAction
Dragon
-protects: Retrievable*
+getCharacter(): 'D'
#act(x, y): EnemyUpdateAction
+beAttacked(int): Pair<int, int>

Design (& Design patterns)

- Map Generation
    - To group the tiles into chambers, tiles were labelled with a chamber ID by crawling each chamber in a DFS-like manner to find all walkable floor tiles which are adjacent
    - To randomly generate items evenly in tiles:
        1. Shuffle the tiles of a chamber
        2. Randomly pick the type of item spawned based on a distribution arrays
        3. Pop the next pre-selected random chamber ID from the queue of chamber IDs
            a. Use a queue so we do not overfill small chambers (a chamber will show up in the queue at most n times, where n is the size of that chamber)
        2. Get the next tile in the iterator of that chamber ID (recall the tiles were shuffled)
        3. Repeat steps 2 and 3 until there is a free tile (the only case where there isn't a free tile is when the dragon associated with a barrier suit/dragon hoard was placed)
        4. Add item to tile
- Dragons and protected items
    - All retrievables can be protected by an enemy. This is used by dragons to deny access to their hoard, whether it's a barrier suit or a high value treasure pile.
- Merchants
    - Merchants are special characters that, upon construction, create a special Treasure "merchant hoard" in their carrying to be dropped on death.
    - When they die, ownership of that Treasure is granted to the tile they died on.
- Potions
    - Potions are generated on the floor and can be interacted with. When the player attempts to interact, the potion applies its effects to the player then the game deletes the potion.
- Enemy Droppable
    - Enemies are assigned droppable items upon generation
        - The compass is generated without a location. Its pointer is stored in a random enemy as a droppable.
        - Merchants are generated with Merchant Hoards as droppables
- Tiles
    - Tile types, such as "Stairs" and "Vertical Wall" are stored as Enums. A tile struct stores its underlying map tile, as well as optional pointers to an enemy, player, treasure, or interactable

Changes from DD2

Player

- Fields `suited`, `attackMod`, `defenseMod`, and `gold` are protected instead of private
- Methods `playerMove`, `playerAttack`, and `playerPickup` return a boolean, indicating if the player is still alive after that move
- Storing electrum (half a gold) instead of gold to avoid integer division (Orcs retrieve half a gold for every gold)
- Getter functions for `health`, `attack`, `defense`, and `electrum`
- Has a overridden attack function different from that of ***Character***
- `player->pickupGold` can just take an amount, no need to pass `Treasure`
- `player->pickupSuit` needs no arguments, the suit will just call it when it's picked up, and now the player is suited
- `player->infoPannel` displays current information about the player to an ostream

Retrievable

- `onRetrieve` returns a bool to denote whether the item was successfully retrieved. If it was, it should be removed from the board, otherwise it should persist (added to ensure that treasures protected by dragons can not be retrieved before the dragon is defeated)

Enemy

- New protected method `Enemy::walk` for more code re-use
- Enemy does not return a string, instead enum elements for cardinal directions or attack
- `beAttacked` is virtual
    - `Dragon` can override it and "unlock" the protected item when it dies
    - All ‘Merchant’ become aggressive once attacked
- `goldValue()` returns how much gold a particular enemy is worth, so dragons and merchants can be worth 0
- `age` is used to synchronize updates
- Splitting `Enemy::update` into `Enemy::update` and `Enemy::act`, so act can be overridden (merchants are not aggressive by default)

Dragon

- Override `beAttacked` to "unlock" its protected item when it dies

Treasure

- Returns 'G' not 'B'

Tile

- Overload for output stream operator
- Added chamber ID
- Added separate `movablePlayer` method
- Added `x`, `y` location to tile

Compass

- When the `Compass` is retrieved, the player doesn't need to know, the board does
- Compass is owned by an enemy

Game

- `Board` is a friend class of `Game`
- `Game` needs a private method `nextLevel`
- `render` becomes `update`, a private method to update board, render board, then render bottom text
- `render` is needed to render without side-effects (initial display)
- Added a pointer to `Player`
- All player interaction functions should return whether the player is still alive
- Added private static helper method `randomPopulateMap`
- Added private static helper method `labelChambers`
- Added private static helper method `traverseChamber`
- Added field `shouldGenerate`

Dragon Protected

Board

- playerLocation is also useful for enemies checking whether they can attack the player, open it up as read only through a method `std::pair<int, int> getPlayerLoc()`
- Need to add an `Update()` function which updates all enemies on the board
- Needs a private helper function to find the tile offset by a `CardinalDirection` from a particular x, y coordinate
- Adding `stairLocation()` so enemies can know to avoid stepping on the stairs
- Added private vector of sets of coordinate pairs `chambers`

Character

- `Player`s don't care about board, only `Enemy`, so move `Board&` to `Enemy`
- Add `getDefense` and change getPower to getAttack so the player can override to use attack/defense mod

Potion

- Has a type describing itself, as well as what it will do to the player upon interacting with it.

Resilience to Change

- Max Levels is set as a variable to account for when the user wants to add levels
- Random generation is generalized to account for when users want to add more chambers or a different map
    - Chamber locations are determined by an algorithm
- Highly extensible template pattern overrides allows enemies to act in unique ways. They can explore the map around them and interact with the player at a distance through act and the board which they have access to.

Answer to Questions

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?
    a. Have a base `Player` class, and different races inherit from that class. The `Player` class stores `hp`, `atk`, and `def`, and other important values.
    b. Other overrides:
        i. `Elf` class overrides the `usePotion` method to reverse all potion effects by taking the absolute value of their effects
        ii. `Dwarf` overrides the `pickUpGold` method by doubling the amount of gold picked up, and `Orc` halves it
        iii. `Human` overrides `calculateScore` so it can 1.5x the number.
2. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?
    a. Yes, generation for `Enemy` should be different from generating `Player`. Although both `Enemy` and `Player` inherit from the same `Character` class, `Player` is owned by `Game`, whereas `Enemy` is owned by `Board`.
    b. The properties of `Player` are chosen by the user. The generation of `Player` should only involve randomly generating the starting coordinates when `Board` is generated, such that it does not overlap with an `Enemy`.

    c. Multiple `Enemies` are randomly generated when `Board` is generated, and their types and properties are also randomly generated (their properties are determined by their type).
3. How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?
    a. Template method pattern, each subclass of enemy implements can override the `act` method to have special actions.
4. 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?
    a. Decorator pattern would work well for this, however we have opted to include integer fields `attackMod` and `defenceMod` which we just add as a factor to `attack` and `defence`, and directly modify the health of the player for health potions (bounded by 0 and max)
5. How could you generate items so that the generation of Treasure, Potions, and major items reuses as much code as possible? That is for example, how would you structure your system so that the generation of a potion and then generation of treasure does not duplicate code? How could you reuse the code used to protect both dragon hoards and the Barrier Suit?
    a. By generalizing the items on each floor tile in a `Tile` struct, we can have a single source of truth for the game state. We can call a common `getCharacter` method on treasure, major items, and characters, orTile, to get the character of whichever object is present.
    b. Treasure and major items are `Retrievable` and are retrieved by walking over them (if they have no protector).
        i. `Compass` overrides the `retrieved` method by showing stairs when retrieved
        ii. `BarrierSuit` overrides the `retrieved` method by updating a player field `suited`
        iii. `Treasure` updates the `retrieved` method by adding to the player's gold count
    c. Potions are `Interactable`, and are different because the player needs to explicitly ask to interactive with one
        i. Different potions just modify different attributes of `Player`. Encode these as integers such as `attackMod` and `healthMod`, and just use arithmetic on them.

Final Questions

1. 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?
    a. Always read the guidelines

    b. It's easier to work together with real-time collaboration software such as vs-code liveshare, as many functions depend on each other to be implemented and abstracted.
    c. Working synchronously from the same location was by far more productive, because even though we bantered more than otherwise, we were more focused and energized overall.
2. What would you have done differently if you had the chance to start over?
    a. We would have read over the specifications and made a summary, as most of our issues were from not remembering the long specifications.