CC3K Project Plan of Attack

1. Responsibility Breakdown

- **Jimmy**: Game loop, command interpreter, I/O, floor generation, final integration
- **Lydia**: Item class, subclasses Potion and Gold, Enemy AI logic
- **Sian**: Characters, combat, potion decorators, interaction logic

2. Project Breakdown and Timeline

July 15

- **All**: Set up Git repo and environment
- **All**: Finalize initial design and UML diagrams for characters and items

July 16-19

- **Sian**: Implement base Character class, and subclasses Player
- **Lydia**: Implement Item class, subclasses Potion and Gold
- **Jimmy**: Implement board rendering and cell representation (Cell, Chamber, etc.)

July 20-21

- **Sian, Lydia**: Implement combat mechanics (attack, damage formula, initiative, enemy hostility rules)
- **Sian**: Implement potion effect application and item collection logic
- **Jimmy**: Implement command parser and basic turn loop (movement, attack, use item)

July 21-22

- **All**: Work on random generation (floors, chambers, characters, items)
- **Jimmy**: Implement map loading from file (with potion/gold/enemy codes)
- **Sian, Lydia**: Add special behavior for races

July 23-24

- **Jimmy**: Integrate systems and test game loop
- **Sian, Lydia**: Implement all enemy behaviors and floor transitions

July 25

- **All**: Full playtesting, edge case checking, polish
- **All**: Prepare submission package and documentation

3. Design Question Responses

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?
We establish an abstract Player base class and concrete subclasses for each race (Shade, Drow, Vampire, Troll, Goblin), each overriding methods like attack(), onKill(), or endTurn() to encapsulate its special behavior. In our character-creation module, we simply look at the user's chosen race ID and directly call the constructor of the matching subclass (e.g. new Drow(), new Vampire(), etc.). This centralized instantiation logic keeps the game loop free of if/else ladders.

To add a new race, we define a new subclass of Player (implement its unique overrides) and add one line in the creation switch (or map) to instantiate it. No other part of the codebase needs touching, so extension is trivial.

Question. How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?
Enemies derive from the Character base class, with each type (Halfling, Elf, Orc, etc.) as its own subclass. In the spawn logic we randomly pick a type and then invoke that subclass's constructor directly (e.g. new Halfling()). The actual instantiation follows the exact same inheritance-based pattern as for the player. Both rely on polymorphism and a single place in code where we map an ID or random draw to a new call.

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.
Yes—the very same. We put default behavior in the Enemy base class, then each subclass overrides only the methods that need special handling:

- Halfling overrides beAttackedBy() to apply the 50% miss chance.
- Elf overrides attack() to perform two strikes (unless the target is a Drow).
- Orc overrides attack() to add bonus damage when facing a Goblin.
  This keeps all logic local to each subclass, avoids big switch statements, and leverages dynamic dispatch through inheritance.

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?
We apply the **Decorator Pattern** on top of our inheritance hierarchy:

- Each time a potion is drunk, we wrap the Player reference in a new decorator object (e.g. AtkBoostDecorator, DefWoundDecorator).

- Each decorator overrides methods like getAtk() or getDef() to modify the stat, then delegates to the wrapped component.

- When the player moves floors, we simply drop all decorator layers and continue with the plain Player instance.
  This cleanly separates temporary effects from the core character class.

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?
We define an abstract Item base class with shared interfaces (e.g. use(), display()) and have Potion and Gold both inherit from it. In the item-generation module, we inspect an item type code or perform a weighted random selection, then directly instantiate the correct subclass (e.g. new Gold(), new Potion()). Any common logic—such as random rarity tiers or on-screen placement—is implemented once in Item or in helper functions, so adding a new item type only requires creating a new subclass and adding one instantiation case in the generator.