Cs246: Final Project CC3K
Initial Report

Donald Shi
David Wu
Kevin Zhang

Plan of Attack:

This project aims to implement a functioning game of CC3K by next Tuesday, keeping around 3 days to debug and polish the code, to turn in a well-designed implementation by Friday (July **25th, 2025**). The overall implementation of the game will be **object-oriented**, applying the **design patterns** covered in the lectures, such as observers, decorators, and factories, among others.

The game involves multiple files with relationships such as **composition, aggregation**, and **inheritance** being core themes to link modules together. Classes such as Characters, Items, Floors, and GamePlay will be coded by different team members with separate responsibilities, and the ownership is carefully defined to avoid memory leaks through double-free or incorrect dependencies between files. We also aim to use smart pointers, using unique pointers wherever applicable in the program. Team members will collaborate and communicate promptly to update each person's status and help out whenever / however needed.

The team members are Donald, David and Kevin, with the tasks currently broken down as:

- Donald: Character (PlayerCharacter) system, Combat & Movement interactions:
    - Character Base class
    - Movement Validation
    - Position Updates
    - Populating Enemies rule (Factory Design Pattern)
- David: GamePlay, Game Set Up, Board Display, and connecting files to ensure fluency & **high cohesion** and **low coupling**.
    - Implement the GamePlay class to manage the game loop, I/O handling, floor Generation, and all Board Display.
    - Floor Generation
    - Connecting Floor Transitions and Game conditions (In progress, ended)
    - Ensure Integration checkpoints with other files.
- Kevin: Map, Item, Character(Enemy), Chamber, Tiles
    - Basic Enemy Class
    - Basic Item Class (Potions + Gold)
    - Chamber, Tiles
    - Handle Potion use and trace their effects.
    - Any potentially needed additional parts.

Estimated completion dates:

Thursday:

- Finalize the UML structure and class relationship → Initial UML report finalized.
- Finalize and submit the initial report (this document) to Marmoset.

Friday: Start Coding and create the files:

- Donald: Implement Character
- David: Brainstorm and outline the structure of Gameplay and how to handle I/O
- Kevin: Implement Title, Chamber, and basic Item structure.

We will have the weekends solely dedicated to the project implementation. The actual progress on each day may exceed or be a bit shy of the target. But we will attempt our best to follow the below, with the expectation of sticking to it.

Saturday:

Donald: Implement the movement validation and position update system.
David: Finish setting up GamePlay::run(), start(), and handleInput().
Kevin: Add concrete potion/gold behaviour and item spawning.

Sunday:

Donald: Connect the combat feature and damage dealing to characters.
David: Integrate game display and colour programming.
Kevin: Complete potion, potion usage, discovery.

Monday:

Donald: Enemy Generation, Score Calculation, Health Tracking.
David: Focus on features like restarting, handling quit and game termination.
Kevin: Ensure floor grind print, correct population and handle conflicts.

Tuesday: Linking files together, writing the makefile, and attempting compilation.

Wednesday - Thursday: Debugging (if needed):

- Ensure Compilation, and play the game to identify any potential bugs.
- Testing with the provided floor layouts
- Test Combat / Moving scenarios
- Test Floor / Game progression

Friday: A fully functioning, running implementation **must** be ready for submission.

Questions:

2.1 Player Character

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?

Decorator Design Pattern for Character Generation

To achieve flexible and maintainable character-specific behaviors, our system uses the **Decorator Design Pattern** to build player characters. The core abstraction is the PlayerCharacter interface, which defines all common player actions and attributes, such as attack, move, usePotion, and turn-based methods.

The base class CharacterBase provides the race-neutral implementation of all standard behaviors. To incorporate race-specific effects (such as Drow's potency potion, Troll's regeneration, or Vampire's life-stealing), we use a series of specific decorators, each of which inherits from CharacterDecorator. These decorators only override the behaviors affected by race characteristics and forward all other calls to the wrapped PlayerCharacter.

Therefore, adding a new race is very simple: we just need to implement a new subclass of the decorator and rewrite the necessary methods. The rest of the game - including the combat logic, movement, and other common parts - is completely unaware of the specific race implementation, as it only interacts with the abstract PC interface.

2.2 Enemies

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

Enemy Creation Using the Decorator Design Pattern

To generate different types of enemies in the system, we adopted a **Decorator Design Pattern**. All enemies have common behaviors, such as movement and attack mechanics, which are encapsulated in a shared base class. This base class defines a common interaction protocol with the environment and the player character.
The unique characteristics of each enemy (dodge for halflings, double attack for elves, anti-vampire effect for dwarves, and so on) are implemented through specific decorators that extend the base behavior. For example, a halfling decorator wraps an Enemy base instance and overrides its combat behavior to make the player miss chance 50%.

Similarly, a merchant decorator monitors whether the global hostile flag is triggered before deciding to attack. This approach allows us to combine enemy behaviors in a modular and dynamic way without cluttering the base class with if statements or repeated logic. This also makes it easier to add or modify enemy types later in development.

In contrast, player characters are generated based on a one-time character selection, and character-specific behavior is implemented via inheritance (e.g., via polymorphic overrides or property adjustments). Since the player is a single persistent entity with a fixed type, a simpler structure is achieved. Thus, while both systems use polymorphism, enemy generation benefits more from behavioral composition (via decorators) to accommodate diverse and potentially composite enemy traits.

In our system, we designed a Character base class to encapsulate all shared movement and combat behaviors, including features such as move, attack, and take damage. This allows both PlayerCharacter and Enemy to inherit these core functions, ensuring consistent and reusable combat logic for all entity types in the game.
Specifically, for player characters, we make use of the **Decorator Design Pattern** to implement character-specific features. The abstract class PC Decorator wraps a PlayerCharacter pointer instance and overrides key methods (features) such as usePotion, startTurn, and endTurn to trigger additional effects.
Each character, such as Drow, Troll, Shade..., inherits PC Decorator and gives it unique traits. Additionally, using decorators allows us to easily introduce new characters in the future without having to modify any code or duplicate player logic. This makes the system extensible.

Is it different from how you generate the player character?

No, the way we create enemies is similar to the way we create player characters. Although they play fundamentally different roles in the game, they have similarities, so we follow the same construction method.
The player character is constructed at the beginning of the game based on user input. We start with a generic base PlayerCharacter instance that includes common behavior logic such as character movement and attacks. Then, specific race decorators (such as Troll, Vampire, or Goblin, etc.) are dynamically wrapped around it. Each race's own decorator injects the specific race's behaviors, such as potion enhancement, passive regeneration, or reward for gold collection. This design provides flexibility, allowing us to clearly separate race logic and easily add new races without modifying the existing code.

In contrast, Enermy has a similar design concept. Each enemy type, such as Elf, Orc or Halfling, inherits from the common EnemyDecorator class. Since their action logic and attack logic are the same, we can design these as the base of decorators. All movement and combat logic is implemented within this, while special behaviors (such as the halfling's dodge or the elf's double attack) are achieved through the decorator methods in the corresponding subclasses.
In summary, they share a similar design concept, where the same behavioral logic serves as the base and different racial characteristics are used as decorators.

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.

Subclass-Based Ability Overrides

Each enemy type derives from a common **Enemy** interface that defines hooks such as Spawn, Attack. Concrete classes override only the customized methods they need: a **Troll** implements health regeneration, a **Halfling** overrides the corresponding attack method to model its evasion chance, and a **Dragon** binds itself to its treasure hoard and so on. This use of inheritance and polymorphism keeps each behaviour customized to its class.

2.3 Potions

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?

Modelling Temporary Potion Effects with the Decorator Design Pattern

In order to represent the potion effects without maintaining explicit consumption logs, we will employ the **Decorator Design Pattern**. In this approach, each potion effect is encapsulated in a concrete decorator that wraps the base PlayerCharacter instance.

When the player consumes a potion, the game generates a corresponding decorator. Since each decorator holds a reference only to the component it wraps, the system inherently maintains the correct stacking order and duration of effects. Once a potion’s effect expires - determined by battle turns - the associated decorator is simply removed from the chain. When a player drink another potion, the game adds one more wrapper around the raw character.

2.3.2 Treasure

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?

Template Design Pattern for Shared Generation Workflow

We use the **Template Design Pattern** to define an algorithm for item generation. The high-level steps include determining the number of items spawned, choosing an unoccupied tile on the floor, creating the item instances, and registering them on the floor tiles. Subclasses override only the *methods* that provide item-specific behaviour, which decide the type of item generated on the floor.

The pattern also makes the system highly extensible: to introduce a new item category — such as treasure (gold) — we simply implement a concrete factory subclass that defines its creation logic. All the common mechanics for placement and registration are inherited from the base.

The benefits of this structure are **maximum code reuse**, **consistent behaviour**, and **low maintenance overhead**. Fixes or performance improvements to the core generation algorithm instantly apply to all subclasses. By leveraging the Template Design Pattern, we achieve a clean and reusable code that scales naturally across current and future item types.