CC3k+ Project Design

Jian Feng

March 27, 2025

1 Introduction

This document outlines the plan of attack for the CS246 Winter 2025 project, ChamberCrawler3000+ (CC3k+). The plan includes a breakdown of tasks with deadlines, and answers to the choice of implementation.

The game features a modular design that allows for optional DLC content to be toggled through command line flags, providing flexibility in gameplay experience. The main executable supports the following flags:

- **-enableWeather**: Activates the Dynamic Weather System DLC
  - Adds weather effects that modify visibility and movement
  - Integrates with existing effect system
  - Can be toggled without code recompilation
- **-enableQuest**: Activates the Quest System DLC
  - Adds optional side quests with rewards
  - Includes kill and collection objectives
  - Leverages existing NPC and item systems
- **-file ¡filename¿**: Specifies custom level file
  - Allows loading of custom board layouts
  - Supports both default and custom level formats
- **-seed ¡number¿**: Sets random seed
  - Enables reproducible game states
  - Useful for testing and debugging

The base game experience remains unchanged when DLC features are disabled, ensuring backward compatibility and maintaining the core gameplay mechanics. The DLC features are designed to enhance the game without compromising the original experience, providing players with the flexibility to customize their gameplay.

2 Overview

The implementation tackles several key challenges:

1. **Spawn Rate Management**
   - Board uses constants for entity counts: POTION_CNT = 10, GOLD_CNT = 10, ENEMY_CNT = 20.
   - Random tile selection from available floor tiles using `getRandomTile()`.
   - Special handling for Dragon Hoards:
     (a) Dragons are placed adjacent to their hoards.
     (b) Protected item mechanism links Dragon to its hoard.
     (c) Position tracking ensures spatial relationship.
2. **Unique Item Management**
   - Compass:
     - Generated once per floor during enemy generation
     - Assigned to a non-Dragon, non-Merchant enemy
     - Board tracks compass state with compassPickedUp flag
   - Barrier Suit:

- Singleton pattern ensures single instance
- Board tracks suit state with setSuit flag
- Protection mechanism shared with Dragon Hoards

3. **Enemy Special Abilities**
   - Virtual useSpecialAbility() method in Enemy base class
   - Specialized implementations:
     - Vampire: Steals 10 HP from player
     - Goblin: Steals 5 gold from player
     - Troll: Regenerates 5 HP
     - Phoenix: Resurrects once with half HP
     - Merchant: Drops merchant hoard when killed
     - Dragon: Protects its hoard
4. **Enemy State Management**
   - Board maintains vector of enemyTiles for efficient updates
   - Enemy attributes (HP, ATK, DEF) initialized based on type
   - State tracking:
     - Health management with bounds checking
     - Death handling with item drops
     - Protected item relationships
     - Merchant hostility tracking
5. **PC-Enemy Interaction**
   - Combat system:
     - Damage calculation considering ATK and DEF
     - Special ability triggers during combat
     - Death handling with item drops
   - Movement system:
     - Valid move checking
     - Enemy tracking and updates
     - Collision detection

3 Design

The system implements several key design patterns and architectural decisions, with a strong focus on high cohesion and low coupling:

3.1 High Cohesion Analysis

- **Functional Cohesion**
  - Enemy class hierarchy:
    - Base Enemy class encapsulates common enemy attributes (hp, atk, def)
    - Each enemy subclass (Vampire, Goblin, etc.) focuses solely on its unique special ability
    - Clear separation of combat stats and special behaviors
  - Board class responsibilities:
    - Manages tile grid and entity placement
    - Handles movement validation and execution
    - Coordinates entity interactions and combat
    - Maintains game state (compass, merchant hostility)
  - ItemGenerator class:
    - Centralizes all item creation logic
    - Type-specific generation methods
    - Clear factory method patterns for each item category
- **Sequential Cohesion**

  - Board initialization sequence:
    * loadFromFile() loads basic layout
    * createEntity() populates entities
    * handleItemProtection() sets up protection relationships
  - Combat sequence:
    * calculateDamage() determines damage
    * attackEnemy()/attackPc() applies damage
    * useSpecialAbility() triggers effects
    * die() handles death consequences

3.2 Low Coupling Analysis

- **Interface-Level Coupling**
  - Entity base class:
    * Provides common interface for all game entities
    * Allows polymorphic handling in Tile class
    * Reduces direct dependencies between types
  - Effect system:
    * Abstract Effect class defines clear interface
    * EffectManager interacts only with interface
    * Concrete effects remain independent
- **Data Structure Coupling**
  - Tile class:
    * Uses weak_ptr to Board to prevent circular references
    * Encapsulates entity and position data
    * Provides clean interface for Board operations
  - Type system:
    * Centralized Type enum reduces direct dependencies
    * TypeCategories provides grouped functionality
    * Consistent type checking across system
- **Communication Coupling**
  - Factory patterns:
    * EnemyGenerator and ItemGenerator isolate creation logic
    * Board only knows about abstract types
    * Clean separation of concerns
  - Protected item mechanism:
    * Items maintain their own protection state
    * Enemies track protected items independently
    * Board coordinates without direct coupling

3.3 Design Patterns Implementation

- **Factory Pattern**
  - ItemGenerator:
    * generatePotion(), generateGold() methods
    * Type-based creation logic
    * Centralized item instantiation
  - EnemyGenerator:
    * generateEnemy() with type or random generation
    * Consistent enemy creation interface
    * Encapsulated initialization logic
- **Singleton Pattern**

  - EffectManager:
    * Thread-safe initialization
    * Global access point
    * Centralized effect management

- **Observer Pattern (Implicit)**
  - Board-Tile relationship:
    * Tiles observe Board state
    * Board updates trigger Tile updates
    * Weak references prevent memory leaks

From the initial design to the final implementation, my design maintained the original core structure and concept, without refactoring or significantly modifying the original class model. All new functions and features were implemented by adding new methods and fields to ensure compatibility and consistency with the existing design.

4 Resilience to Change

The system is designed to be highly adaptable:

- **Effect System Extensibility**
  - New effects can be added by creating new Effect subclasses
  - EffectManager handles all effect lifecycle management
  - Clear separation between temporary and permanent effects
- **Enemy System Flexibility**
  - New enemy types can be added through inheritance
  - Special abilities are encapsulated in subclasses
  - Protected item system allows for new protection mechanics
- **Item System Modularity**
  - New item types can be added through inheritance
  - ItemGenerator factory methods support easy extension
  - Protection mechanism is reusable across item types
- **Type System Organization**
  - Centralized Type enum for all game entities
  - TypeCategories class for grouping and checking types
  - Easy addition of new types and categories

5 Answers to Questions

1. Answer: How could you design your system so that each race could be easily generated? Additionally, how difficult does such a solution make adding additional classes?
   - PlayerCharacter class is the core class of the player character, which contains basic attributes (such as health `hp`, attack power `atk`, defense power `def`, and gold coin multiplier `goldModifier`), as well as behavior logic.
   - setAttributes(Race r) method dynamically sets the character's initial attributes according to the selected race by a `switch-case` structure.
   - When adding a new race, just add a new type to the `Race` enumeration and add the corresponding attribute modification logic in the `setAttributes` method.
   - This design complies with the open-closed principle (open for extension, closed for modification).
   - Each time you change race, it will reset to default before applying race-specific modifications, ensuring that attributes do not accumulate errors due to multiple switching.
2. Answer: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

- EnemyGenerator class is responsible for dynamically creating enemy instances based on the needs of the game. It uses a factory method or abstract factory pattern internally to determine the specific enemy subclass to instantiate based on the parameters or random numbers passed in.
- While both enemies and player characters are game characters, there are some key differences in how they are generated:
  - Enemies usually need to support a large number of variants to increase the richness and challenge of the game, while player characters focus more on personalized customization options.
  - Enemies are dynamically generated according to preset templates, especially when generating random events or levels. In contrast, player characters are usually carefully configured before the game starts.
  - Enemy instances have a short lifecycle and will be destroyed when the battle ends; while player characters are core objects throughout the entire game process, and their status requires long-term maintenance.
  - Since there be a large number of enemies in the game, their generation process must be efficient and easy to batch process. For player characters, although they also need to be optimized, due to their limited number, the performance requirements are relatively low.

3. Answer: How could you implement special abilities for different enemies. For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?
   - By creating a base Enemy class, and having each enemy with a special ability (such as goblins, trolls, vampires, etc.) inherit from this base class, and then specifically implement each special ability in the subclass. Define a virtual method (such as useSpecialAbility) in the Enemy base class, which will be implemented by the specific subclass to provide specific behavior. For example, a vampire can steal health from the player and increase his own health; a goblin can steal the player's gold; and a troll can restore its own health. This design makes it very simple to add new enemy types to the system. Just create a new subclass and implement the useSpecialAbility method.
4. Answer: 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 Effect class defines an interface that all concrete effect classes (such as BoostAtkEffect) must implement. It includes methods for applying effects, removing effects, and checking whether it is a temporary effect. The EffectManager singleton class is responsible for managing and tracking all effects currently applied to the player character. The clearTemporaryEffects method of the EffectManager is called every time the player enters a new floor to clear all temporary effects. In this way, we avoid directly tracking the specific potions consumed by the player, and instead dynamically apply and remove effects through the decorator pattern.
5. Answer: 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?
   - To ensure that the code for generating gold, potions, and main items is reused as much as possible while avoiding duplication, we use several strategies in object-oriented design: inheritance, interfaces, and factory patterns. Define an Item base class or interface, and all spawnable items (such as gold, potions, treasures, etc.) derive from or implement this base class. This allows us to use a unified way to handle the generation logic of different types of items. Create a specific class for each specific item type, such as Gold, Potion, Treasure, etc. Each class is responsible for its own characteristics and behavior, but all follow the same interface or inherit from the same base class. Use factory methods or abstract factories to generate item instances. The advantage of this is that the item generation logic can be centrally controlled, and it is easy to add new item types without affecting existing code. Whether it is a dragon's treasure or a barrier set, a common protection mechanism is implemented by adding a protection status property to the Item base class. Different items can set this status as needed.

6 Extra Credit Features

We propose the following DLC features that can be toggled without code recompilation:

- **Dynamic Weather System DLC**
  - **Feature Description:** Weather effects that modify gameplay mechanics
  - **Implementation Approach:**
    - Created WeatherEffect class implementing Effect interface
    - Added weather types (rain, storm, fog) affecting visibility and movement
    - Integrated with existing EffectManager
    - Implemented weather generation through generateWeather() function

    - Added weather effects management in EffectManager with addWeatherEffect() and clearWealtherEffects()
  - **Design Integration:**
    - Uses existing effect system architecture
    - Extends TypeCategories for weather types
    - Maintains current game balance when disabled
    - Activated through -enableWeather command line parameter

- **Quest System DLC**
  - **Feature Description:** Optional side quests with rewards
  - **Implementation Approach:**
    - Created Quest class hierarchy with KillQuest and CollectQuest
    - Implemented QuestManager singleton for managing active quests
    - Added quest tracking in PlayerCharacter (killCounts, itemCounts)
    - Integrated quest updates into game loop
    - Implemented quest generation in Game::initializeQuests()
  - **Design Integration:**
    - Uses existing item generation system for rewards
    - Leverages current NPC interaction framework
    - Optional content that doesn't affect main gameplay
    - Activated through -enableQuest command line parameter

These DLC features are designed to:

- Add significant gameplay depth without modifying core mechanics
- Showcase object-oriented design principles (inheritance, polymorphism)
- Maintain clean separation between base game and optional features
- Leverage existing architectural patterns
- Be easily toggled through command line parameters
- Provide meaningful rewards and challenges
- Integrate seamlessly with existing systems

7 Final Questions

1. Answer: 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?
   - This project reinforced the value of agile development—rapid prototyping, frequent iteration, and a flexible response to change are essential. It also highlighted the importance of collaboration; rather than struggling in isolation, reaching out for help and working together can greatly enhance the development process.
2. Answer: What would you have done differently if you had the chance to start over?
   - If given another chance, I would start by leveraging proven examples and best practices from established projects instead of building everything from scratch. Often, the widely adopted standard approach yields better results than trying to innovate solely based on one's own ideas.