Design Document for CC3K

Matthew Sekirin, Sophie Zheng, Justin Leung

1 Overview

As a game, CC3K requires user input. The entry point of the program is in `main.cc`, which creates a single `CC3K` game instance and to facilitate interactions between the user and the actual game state. It also sets up a text (or graphics) display and attaches it to the game state. The program then outputs a welcome message, and from there, the program enters a loop of continuously waiting for the user to issue input, processing the input, and outputting a new game state.

The `CC3K` game class is the main game controller. The class exposes public methods such as `playerMove(), playerAttack(), playerUsePotion()` to `main.cc` , which are each called when the user issues the appropriate command. The `CC3K` class is responsible for initializing and destroying all parts of the game state (player, gold, enemies, potions, etc.), which is handled by a `DefaultLevel` instance.

Anything that can interact with the game world is derived from the superclass `Entity`. This includes the player, enemies, potions, gold, and the stairway. Furthermore, enemies and player classes inherit directly from a `Character` class, and potions and gold inherit directly from an `Item` class. The `CC3K` class handles the interactions between entities in the game world (e.g. enemy movement, combat, picking up gold, etc.), entering the next level (when player moves onto a staircase), and checking the winning and losing conditions (e.g. if player dies from enemy attacks).

2 Design

2.1 Representing things inside the game world: `Entity`

CC3K is a game with a top-down view on a 2D plane. Each “thing” in the game can thus be described by their x and y coordinates, as well as a symbol (a char) that represents it. We thus create an `Entity` class to model anything that resides *within* the game environment that has a position in space, which includes players, enemies, potions, treasure, and the staircase which derive from it. Player and enemy movement is implemented by calling the public position setter functions that inherit from `Entity`. Figure 1 shows the UML diagram of Entity and its immediate subclasses.

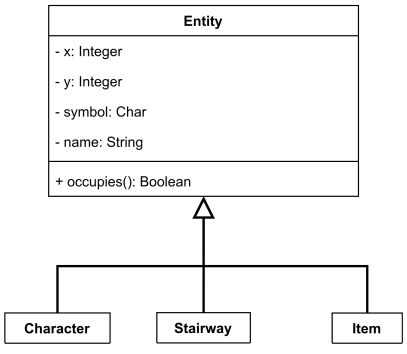

Figure 1: A simplified UML diagram showing the Entity class and its immediate subclasses (with their methods and attributes omitted).

2.2 Representing the game environment: `Floor` and `Cell`

The environment is represented by a single `Floor` instance which defines the floor map. The `Floor` class owns a 2D grid of `Cell's` which make up the game board. Each `Cell` is either a walkable tile, wall, passage, or door. The `Floor` class provides a public method `chamberAt(int x, int y)`, which specifies the chamber number that the x and y coordinates represent, with -1 being no chamber. The `CC3K` game class uses `chamberAt` during level generation to ensure that all entities have equal probability of spawning at a given chamber (i.e. spawning probability is not dependent on chamber size), as well as other logic such as ensuring that the player does not spawn in the same chamber as the staircase.

2.3 Level generation (Factory design pattern)

Level generation is done in accordance with the Abstract Factory Design Pattern. All characters and items are created by an instance of the `DefaultLevel` class that is contained within `CC3K`. This class is an implementation of the `Level` abstract factory class that declares virtual methods for generating all characters and items. In particular, this includes: `generatePlayer()`, `generateStairway()`, `generateGold()`, `generatePotion()` and `generateEnemy()` (not including parameters). This approach abstracts the generation code from the client. The final method that is part of the core features of the game is `placePlayer()`. This is useful since the same player is present throughout different floors, with a random position at each floor. Instead of re-creating a player object, a call to this method allows a new random position on the floor to be established based on the logic in the concrete factory class being used.

2.4 The display (Observer design pattern)

To implement the text-based display for the game, we applied the Observer design pattern. We accomplished this by making the `CC3K` game class a concrete subject that subclasses the `Subject`. When the game starts, we attach a concrete observer `TextObserver` to the `CC3K` subject. The `TextObserver` implements a `notify()` method that loops through the width and height of the game board and queries the state of the game using a method `getState(int x, int y)` that is exposed by `CC3K`, which returns the symbol representing the entity occupying the cell at the position, or the cell itself if it is not occupied. This observer also gets the game status and messages that it displays underneath the board to show the player's statistics and recent actions that occurred. After every turn, the game calls the `CC3K` method `notifyObservers()` , which in turn calls the `TextObserver`'s `notify()` method, thus rendering the board and the user interface.

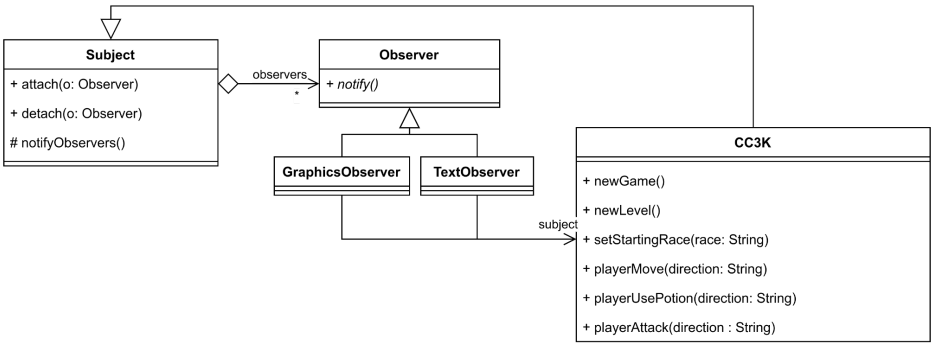

Figure 2: UML diagram of the Observer design pattern applied to CC3K.

2.5 The Player

The `Player` class is a subclass of the `Character` class and thus inherits members from it (Atk, Def, HP, move, attack, etc.). We define subclasses of the `Player` class, each representing a player race (`Drow/Shade/Goblin/Troll/Vampire`). The Player class defines a “default” attack method which implements the damage formula in the specification document. Specific races can override the default attack logic as necessary - they can call the superclass attack method and add additional behaviour (e.g. Vampire attacks as normal, but also gains +5 hp after), or completely rewrite the logic. In addition to the inherited attributes from `Character` and `Entity`, the `Player` also defines some private attributes such as `startingHp`, `startingDef`, `startingAtk`, which are used to cap the maximum HP when applying potions. Furthermore, we define `potionMagnifier` which defines the amount by which a potion’s effects are magnified by. By default it is 1.0, but the `Drow` subclass overrides this value to 1.5.
This class also exposes public methods that are used by the `CC3K` game class which define the expected player behaviour - `applyPotion()` to allow the player to consume potions and `applyPermanentPotions()` which removes temporary potions (e.g. WA, WD, BA, BD) and keeps permanent potions (e.g. BH, PH) and is called when a Player enters a new level. To implement potion effects, the class stores a vector of (shared pointers to) Potions. When the player picks up a potion, we store the potion in this vector and apply the effect, and when the player enters a new level, we remove the potion from the vector if it is a temporary potion or keep it otherwise.

2.6 Enemies

To implement enemies, we create an `Enemy` class that subclasses the `Character` class. We create concrete enemy clases which subclass `Enemy`: `Human, Dwarf, Halfling, Elf, Orc, Merchant, Dragon`. Each concrete subclass defines the respective HP/Atk/Def attributes required of any `Character`. On each turn, the game checks if the enemy is in range of attacking the player using an `inRange(Character defender)` method, which by default returns true if the defender is one tile adjacent to the enemy.

Special enemy behaviour is thus implemented by overriding the default methods. For example, to allow the dragon to attack the player if they are nearby the dragon hoard (not the dragon itself), we override `inRange` to extend the range to the dragon's hoard as well. We can also override `attack` to apply special behaviour (e.g. add a damage multiplier if the player is a certain race). The `Dragon` is also special in that it "has" a `DragonHoard` (i.e. an aggregation relationship), which is expressed through a smart pointer attribute `theHoard` in the `Dragon` class.

2.7 Potions and Treasure

To implement treasure piles, we define a `Gold` class that inherits from the `Item` class. The `Gold` class has two attributes: an integer `value` representing the amount that the pile is worth, as well as a boolean `pickup` that represents whether the pile can initially be picked up (true by default). To implement different types of gold piles, we define concrete subclasses `SmallPile, NormalPile, MerchantHoard,` and `DragonHoard` (cannot be picked up unless the associated dragon has been slain).

To implement potions, we define a `Potion` class that also inherits from the `Item` class. Each potion has a boolean attribute `permanent` which is true if the potion's effects remain after the player enters a new level and false otherwise. Furthermore, `deltaHp/deltaAtk/deltaDef` are integer attributes that define the player's change in HP/Atk/Def after consuming the potion. The concrete potions (RH/PH/BA/WA/BD/WD) each subclass the `Potion` class with the appropriate attribute values as prescribed by the specification. Whenever a player consumes a potion, the CC3K class is responsible for attaching the potion (and its effects) to the player and removing the potion from the game world.

3 Resilience to Change

From the start of the project, we prioritized resiliency to change and extensibility as core objectives. By doing so, this allowed us to implement extra credit features on top of our base game easily.

We applied all four OOP principles throughout the codebase to ensure that it is easily understandable, error-free, and highly extensible. We applied **inheritance** to prevent code duplication and allow for easy extensibility: for example, all enemies attack according to a default (virtual) `attack()` function inherited from `Enemy`, and if required to create a new enemy type with a special attack, we simply create a new class that derives from `Enemy` and overrides the default attack behaviour. We apply **polymorphism** in parts where core game logic does not change and we can treat entities uniformly: for example, the CC3K game class is able to store a vector of `Enemy`'s (rather than storing separate vectors of `Orc, Human, Dragon,` etc.) and we can move all enemies in that vector enemy is guaranteed to provide a `move()` function. We applied **encapsulation** by defining all member attributes to be private by default and only providing getter/setter methods for attributes that we expect to change (e.g. x/y coordinates). Lastly, we applied **abstraction** through the CC3K game class by providing exposing a simple interface (using public methods) which the `main` function calls after receiving user input (e.g. if user inputs "we", `main` will call `game.movePlayer("we")`, if user inputs "u so" then `main` will call `game.usePotion("so")`), with the actual implementation details abstracted away. As new logic and features are added, the existing code does not need to be changed. At the same time, the design allows for common functionality to be grouped together, abstracted, and reused to yield a more simple and intuitive implementation.

To accommodate for potential change we also applied design patterns whenever it was beneficial to do so. These helped ensure **low coupling and high cohesion**, with each pattern used having a specialized purpose. If new game features are added or existing ones are edited, it is sufficient to simply add another class or another method as long as it fits into the general design pattern. This ensures minimal changes need to be made and makes it clear what part of the code is responsible for what task. For example, in designating the Observer design pattern to handle all screen rendering, we ensured low coupling as the concrete observers were solely attached to the CC3K game class (the concrete subject) and we ensured high cohesion as all of the relevant rendering code was self-contained within the concrete observers and nowhere else. To add a graphical display in addition to the textual one, we simply created a new `GraphicalObserver` class that followed the *Observer*'s interface and added it to the game. Since it was used in exactly the same way as the `TextObserver` by the game class, no additional changes had to be made. A more subtle benefit of the use of design patterns was its positive impact on our team's collaboration. All team members had a common understanding of any given pattern and its purpose, and so could verify the implementation correctness against the course notes. This also meant everyone could understand different parts of the code without having necessarily written it, and could add new features relating to any part of the game more efficiently.

In addition to the above, we also incorporated well-defined constants. This enables us to easily change features of the game to accommodate extra credit or user requested features. For example, if we want to change the number of potions or enemies that spawn, we would only need to change `NUM_POTIONS` or `NUM_ENEMIES`, instead of changing multiple lines because of hard-coded numbers.

In conclusion, by using OOP principles, design patterns, and well-defined constants, we were able to make our program resilient to change. It is easily maintainable if any failures happen, and easily extendable as well.

4 Answers to Questions

Q1: 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?

First, we can design our class hierarchy to contain the abstract superclass `Player` from which all player races derive from. This means that for each specific race such as `Drow`, `Vampire`, etc. we will have a concrete class deriving from `Player` that will contain the relevant characteristics of the race. Generating an object of a specific race would then require no parameters in the constructor call, since it would be the concrete classes that would manage their own behaviour and attributes. In particular, any shared characteristics such as hp, attack and defense would be moved upwards in the hierarchy, and the concrete classes would only be supplying these values in their call to the `Player` class constructor.

On top of this, we could also design our system to use the factory (or abstract factory) design pattern to make generating races easier. This would mean that this code would be encapsulated in a factory class, and all that would have to be done to create a player would be to call the `generatePlayer` method of the concrete factory class instance. This class could deal with details like creating a `Player` of the appropriate race and setting its position on the map using our `Floor` class methods.

The combination of these approaches makes creating additional races relatively straightforward. For example, if we wish to allow users to create their own customized race, a straightforward way to implement this would be to create a `CustomRace` class deriving from `Player` which would allow users to specify the attributes of `Player` via standard input. Additionally, any methods relating to player generation would have to be updated to accommodate for this introduction in the abstract and concrete factory classes. While this may seem like a lot, it should be kept in mind that player generation is not overly complicated and only one concrete factory class should be required for the base features of player generation. Unless we change the logic of how the new race is being placed on the map or add a new concrete factory class, there will only be a couple of small additions of code outside of the new race class.

Q2: How does your system handle generating different enemies? Is it different from how you generate the player character? Why or why not?

We create an abstract superclass `Enemy` to model different enemies. We can apply the Factory design pattern to generate different enemies, similarly to the generation of a player. Generating enemies is slightly more complex than generating a player. As specified in the requirements document, generating enemies comes after all other objects (player, potions, stairway, gold) have been generated. So for each enemy, we must check if the generated position collides with anything else we have already generated. Since we generate the player first, we do not need to perform this check. In other respects, determining the position of an enemy or a player should be the same, and this portion of the code will be reused. Another difference is that we will need to store all the enemies in a vector to keep track of all enemies. This simply means adding new enemies to the back of the vector when they are being generated in the proper amounts.

Since enemies and the player character share similar attributes, the `Enemy` and `Player` classes will both derive from an abstract superclass `Character`, which will hold shared attributes such as HP, Def, Atk, and movement. All data relating to a character is encapsulated in either the `Character` base class or these subclasses. Thus, there is no need to pass in constructor parameters for either a player or an enemy in the factory class.

Q3: How could you implement the various abilities for the enemy characters? Do you use the same techniques as for the player character races? Explain.

We can define virtual methods for the `Character` class which specify the special abilities of each player race or enemy type. For example, we may have an `abilityPassive()` method which by default does nothing (most races do not have any passive ability), but the `Troll` class can override this method to increase its HP by 5, in order to model the Troll's special ability that it gains 5 HP every turn. Similarly, we can implement `abilityOnAttack(Character defender)` to model special attack abilities by player classes against other enemies (e.g. Orcs do 50% more damage to Goblins) and `abilityOnKill()` to override the default behaviour when a player class kills an enemy (e.g. Goblins gain an extra 5 gold for every kill).

Since these methods can be declared in the `Character` class, this implementation will allow abilities to be applied on players and enemies without any code duplication. Players and enemies share a few similar behaviours (e.g. a passive ability by simply existing every turn, or an attack ability), and since

enemies cannot kill other enemies, an `abilityOnKill` function does not need to be implemented for enemies but only for player race subclasses.

Q4: The Decorator and Strategy patterns are possible candidates to model the effects of potions, so that we do not need to explicitly track which potions the player character has consumed on any particular floor. In your opinion, which pattern would work better? Explain in detail, by discussing the advantages/disadvantages of the two patterns.

The Decorator design pattern would better model the effects of potions than the Strategy pattern. The Strategy pattern defines and encapsulates a family of algorithms, and allows for interchangeability between a set of defined algorithms. The Decorator pattern allows for adding functionality to an existing feature (in this case, the player state).

When considering how potions typically work in rogue-like games, potion effects *stack*, that is, it is possible for the player to consume an Atk Boost potion followed by a Def Boost potion and expect that their Atk and Def *both* increase by 5. The Decorator pattern could model this by maintaining a linked list of potions that the player has consumed, and traversing the linked list and incrementing the player state with the relevant effect associated with that potion. In contrast, the Strategy design pattern would only allow the player to switch between the given "effect algorithm" but not allow for more than one effect at a time (e.g. the player can only possess a +5 Atk **or** +5 Def boost, but not both). Thus, the Decorator pattern most naturally models the intended "effect stacking" behavior of potions, and the Strategy pattern would be better suited for game features with distinct single-selectable options (e.g. the current weapon that the player is holding).

Q5: 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 can create an abstract superclass `Item` from which the `Treasure` and `Potion` subclasses derive from. These subclasses will override a public method `activate(Player player)` which will alter the player character's state when the player picks up the item (e.g. increase the player's gold count if it is treasure, or apply an effect if it is a potion). The `Item` class is a subclass of the `Entity` abstract superclass, so it will always contain x and y coordinates on the floor as well as a displayed symbol (which will be specified by the constructor); potions have a `P` symbol and all gold will have a `G` symbol.

5 Extra Credit Features

See demo document for demonstration and instructions for each of these extra credit features.

Graphical display: We implemented pixel graphics in our game, giving it a graphical display. We used the `libxpm` library which allows for drawing ".xpm" images (X PixMap) which support RGBA. To solve this problem with good OOP design, we extended upon the **Observer** design pattern which we already applied in the base game. The base game uses a `TextObserver` class as a concrete observer, and the CC3K class is a concrete subject. To add graphics, we created another concrete observer: the `GraphicsObserver` class, which overrides the `notify()` function to display graphics by communicating with the Xwindow API. Whenever the CC3K game renders (and thus notify all observers),

the graphical observer will loop through each cell of the board, get the symbol that is on the board, and display the image corresponding to the symbol (e.g. an image of the player).

Color output: To implement this feature, we created a utility class `Color`, which contains static string constants that represent the colour codes. To colour a string output, the string constant is printed, which acts as an IO manipulator. This `Color` class is used by a Message (a wrapper class that simply contains text and colour attributes), and the TextObserver prints each message with the associated colour.

XP/level system: To incentivize engaging in combat, we add a level up game mechanic, whereby whenever the player kills a sufficient number of enemies, they will gain a level and become stronger. We implemented this game mechanic by expanding the `Player` class to have extra `exp` and `lvl` attributes and a `levelUp()` function that will permanently increase the player's maximum HP / Atk / Def. Whenever the player kills an enemy, they gain a set amount of XP, and level up when the XP exceeds a certain amount determined by the level.

New enemy with intelligent pathfinding: All enemies in the base game move randomly or are stationary. Thus, we were interested in creating an enemy that was capable of pathfinding (chasing players and avoiding obstacles). To implement this, we created a new enemy called `Pathfinder` which uses a **breadth-first search** (BFS) to generate the shortest path to the player's location, while keeping track of the cells that have been visited. Once the player has been found, the BFS algorithm back-traces the visited paths to determine where to move next. Since the player and other enemies also move after every turn, the Pathfinder enemy must recompute the shortest path after every turn.

Fog: Many roguelike games limit the player's line of sight to only a few cells away and hide all other entities in a fog to make gameplay more challenging. We implemented this by extending upon the TextObserver and conditionally rendering the game state at a given position only if it is a certain distance away from the player's position. Specifically, we render a cell only if the cell coordinates satisfy the inequality of an ellipse: $a(cell.x - player.x)^2 + b(cell.y - player.y)^2 < k$ where a, b, k are constants. This gives the player an ellipse-like/circular field of view.

Buying from the merchant: As suggested as a bonus feature in the specification, we allow players to buy from merchants. This game mechanic functions very similarly to our implementation of player interactions with potions - the user inputs "b [direction" to buy from a merchant in a given direction, and the merchant will give a random (but guaranteed beneficial) potion. All potions cost 3 gold, and the player cannot buy if they have insufficient gold. The merchant will also refuse to sell to the player if they have killed other merchants previously.

New player type - Dragonslayer: The Dragonslayer player class deals immense damage to dragons and has the ability to pick up a dragon's hoard without needing to kill them. It also has the highest base HP, Atk, and Def stats in the game. The `Dragonslayer` class derives from `Player` and overrides `attack` for its special abilities, and we add extra logic in the game class to support the Dragonslayer's dragon hoard stealing.

Use of smart pointer and STL containers, no memory leaks: We do not manually manage memory, nor do we use arrays and instead use a variety of STL containers such as vectors, pairs, and queues. To express ownership (e.g. the game class owning all entities, the dragon owning the dragon hoard), we use smart pointers only. We thoroughly tested our game to verify that it does not leak memory.

6 Final Questions

FQ1: 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?

We employed standard software engineering practices throughout the entire development process. The team collaborated on a Github repository where the code was hosted remotely. Instead of pushing all code directly to the main branch, we made code changes on separate branches and then opened pull requests when the changes were ready to be merged onto the main branch. Through this method, other team members can check each others' changes easily. We also used Github's code review features (e.g. diffchecker and code highlighting / discussions) to properly review important pull requests before merging them, which would reduce the likelihood of unexpected behaviour or bugs being introduced into the program.

In addition to the use of branching and pull requests, we learned to set up a rudimentary continuous integration (CI) pipeline using Github Actions. This pipeline was set up to automatically run on each pull request opened as well as every merge to the main branch, and it compiles the program to ensure that no code was accidentally pushed which would fail compilation, or if there was a mistake in dependencies.. The use of CI in our project helped verify that incremental changes were of high quality and stable.

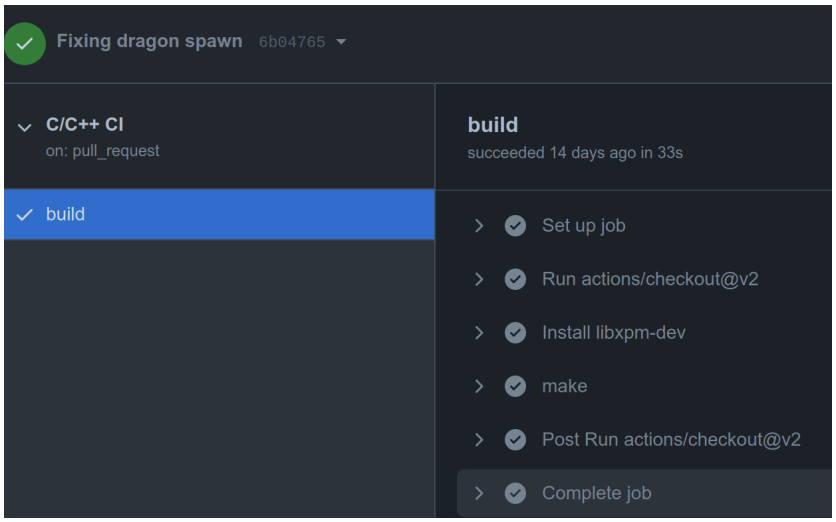

Figure 3. Example CI pipeline job running on a Git commit, implemented using Github Actions.

Furthermore, we learned how important communication is. We set up weekly calls to check in on everyone's progress, and set up goals for the next week. This way, we were able to complete the project early and have extra time to test and implement extra features. We also considered project management software such as Jira and Asana. However, since the project was relatively small and over a short timeline, we ultimately decided that such project management tools were unnecessary.

Overall, we all learned a variety of both technical and soft skills that will help us in the future when working on professional software development teams.

FQ2: What would you have done differently if you had the chance to start over?

In terms of design and planning, we were satisfied with how early we started, but we would have benefitted from more effort put in designing. In particular, we noted that the application of some design patterns (e.g. decorator pattern for potion effects) were more complex than anticipated, and so we opted for a simpler implementation at the cost of extensibility and abstraction. However, we were satisfied with the degree to which we applied other design patterns as well as our overall design, with the time given.

One technical issue that we did not anticipate was the growing complexity of the CC3K game class. Since the game class is responsible for handling all interactions between the player and enemies, potions, gold, and the staircase, the implementation of all of the logic made the class definition very lengthy. This issue was partly alleviated by the use of the Factory design pattern, which abstracted away the level generation code into a separate `LevelFactory` that the CC3K class owns. Despite this, the game class was still complex, and handling the various special cases described in the specification added a lot of extra conditional logic. A potential solution that we discussed was to create "game mechanics modules" which are abstract classes such as "`Movement`" and "`Combat`" that the game class would own and use for each respective game mechanic by passing the appropriate data (e.g. enemies and players). Unfortunately, we did not have enough time to pursue this refactoring.

Some aspects of the game are inefficient and could be easily optimized. For example, in our algorithm to move enemies, we must check each enemy's position against another enemy's position (since we store enemies, potions, and gold in flat vectors) which leads to an $O(n^2)$ time complexity. Since the number of total entities is small (~40), this inefficiency is negligible for the given specifications, but could become costly if we wish to extend the game to spawn more enemies. Some possible solutions to this inefficiency could be to instead store all enemies in a hash map or 2D vector with the x/y coordinates as the keys/indices, so that the collision checking can be done in $O(1)$ time. Furthermore, in our text and graphics rendering, we could make further optimizations by rerendering only entities that have changed positions.

Therefore, if we started over we would probably discuss the design of our program more thoroughly. We would incorporate design patterns from the start instead of relying on refactoring and spend more time thinking about how to make our program more efficient.

7 Changes from DDL1 Design

Overall, there are few differences between our intended design from DDL1 and our DDL2 implementation. We planned to implement a `CellIterator` class that implements the Iterator design pattern to how we handle the game board's cells but we found that this was not necessary for our implementation. We also planned for the `Item` class to contain an `activate()` function that would apply the effect of the item (e.g. increase gold count or apply the potion effect), but we delegated this logic to the game class as it controls the interactions between the gold/potions and the player. Also, as extra credit features, we added an extra concrete enemy class `Pathfinder` and player class `Dragonslayer`.