Overview:

InputDevice: Abstract class which requires its concrete children to implement getUserInput(), a function which gets input from some source (text file, cin, etc.) and return a **Command**. For CC3K we will implement **CinInputDevice** which takes lines from std::cin until a valid command is inputted.

Command: A class consisting of two fields, an Action enum and a Direction enum. **Game** will use the commands and do the appropriate action (move, use, attack, etc.).

Game: This is the controller for our game and holds an **InputDevice**, **View** and **GameModel**. When the game starts, it will continually receive commands from the input device. Upon receiving one, it will apply the command to the GameModel and then render the board using the View.

GameModel: Contains the entire game state including the board (represented as a 2D vector of **Tiles**), player location, stair location, etc. The game model also contains methods for spawning entities, generating floors, etc. GameModel is a child of our abstract **Observer**.

Observer: Subclasses of Observer will observe entities and be notified when an entity dies.

Drawable: Drawables will implement the getChar() method so we can render them. **Tiles** and **Entities** are drawable. The getChar() for a tile will return the character of the entity on top of it or the tile's character if there is no entity.

View: The view renders the game state by taking in a 2D vector of **Drawable** objects which is passed in from **Game**.

Tile: The main way we represent the game state is through the use of tiles. A **Tile** refers to a piece of the map (wall, passage, doorway, floor, etc). A tile is an **Observer** and observes the **Entity** that is on top of the tile.

Entity: Any object in the game that can sit on top of a tile. In our case **Characters** and **Items.** Entities are observed by **Tiles** which are notified when an Entity dies. All entities can die() and have an endOfTurnEffect() which is triggered after the player has moved. Entities will also have a getLoot() function which returns an entity to be spawned when it dies.

Character: A Characters is either an **Enemy** or a **Player**. All characters implement attack() and move() and have atk, def, and hp stats.

Enemy: Enemies will override endOfTurnEffect() to either move randomly or attack the player. Enemies may also hold a pointer to a tile which holds an **Item**. When the enemy dies, that item will be unlocked and able to be picked up. In our case, dragons will hold a pointer to a tile with a dragon hoard or a barrier suit.

Player: Players implement use() so they can use **UseableItems**. All items have a corresponding public method in player (e.g. gainCompass(), applyPotion() etc.). Players have a pointer to **PotionFx** which stores all temporary effects.

Item: All items will implement applyItemEffect which calls the appropriate method on the **Player**.

UsableItem: We organize items into two groups, those that are usable and those that are steppable. **UsableItems** are those that trigger their effect when the player issues a use command on them.

SteppableItem: SteppableItems are those that trigger their effect when the player moves over them.

Design:

PotionFx: In order to represent the temporary potion effects a player has for the floor, we use a **decorator pattern**. When the player uses a potion, a PotionFx decorator is added which holds the attack or defense change that the potion applies. Because we only have one concrete decorator, we do not have an abstract decorator class although it can be added if needed. Each PotionFx has a pointer to either the next PotionFx or null. Clearing the temporary effects is as simple as setting the member holding a PotionFx on the player to null, since we use a unique pointer for this.

Entity actions: Many entity actions require updates to the tiles holding them or the game state in general. As a result, we use a **Subject/Observer** pattern to facilitate such interactions. Entity serves as our subject class as we do not require any other objects to trigger actions in an observer. We have two observer classes: Tile and GameModel. Tile must observe entities to remove them and handle loot generation, while GameModel observes entities so that they may trigger game-wide changes. In our case, the only type of entity that does this is Vampire, since the Vampire class has no knowledge of which enemy has the compass. See the description of our additional features for more information on this mechanic.

Command: Rather than directly calling methods on the GameModel based on the user's text input, we use the **command pattern** by creating a Command object and passing it to the invoker, which in our case is Game. In this way, we provide a separation between the text that the user inputs and the action that we want to perform. If we want to add additional ways to obtain user input, we can create more concrete InputDevices which return Commands to Game.

MVC: GameModel, View, and Game are our **model, view, and controller,** respectively. The GameModel represents our current game state, the Game calls methods on the GameModel and passes the necessary data to the view so we can render the board.

NVI: We employ NVI wherever it makes sense to enforce invariants for a function. For example, die() is a public nonvirtual method for Entity, but within die we also invoke onDeath(), which is private and virtual. This way, every subclass can implement their own functionality when die is called while ensuring that observers are always notified when an entity dies. Moreover, this scheme does not expose unnecessary implementation details in the public interface.

Resilience to Change:

Our design is flexible to changes in the program specifications for the following reasons:

Input Device: Our game takes a pointer to an InputDevice which is an abstract class. We can create various concrete subclasses of InputDevice so that the way that the player inputs their commands can be easily changed.

View: We currently do not use an abstract View but it can be added. We would just need to define the abstract View as any class that implements draw() and takes in a 2D vector of drawables and the other necessary strings. Then, due to our Model-View-Controller software design pattern and the resulting separation of the view, an addition of different graphical interfaces would only require a new concrete subclass of the abstract View. Separating the model view and controller also contributes to our cohesion as each class only handles one aspect of the game.

Entity: Inheritance is a big part of our resilience since the existence of superclasses and subclasses allow for easy extension of superclasses. As an example, our entity class has a virtual method that gets the loot of an entity, which allows the handling of features including compass and merchant hoard drops, and phoenix regeneration. The entity class offers virtual methods with sensible default implementations, allowing for subclasses to modify as much or as little behaviour as necessary.

- **Character:** All characters use common code for attacking and moving. We designate the player and enemies with different teams, with a check in the attack method to prevent a character from attacking another character on the same team.

    - **Player:** The races of players are subclassed within Player so if we were to add a new type of player, we just need to create a new subclass.

    - **Enemy:** The types of enemies are subclassed within Enemy so if we were to add a new type of enemy, we just need to create a new subclass.

- **Item:** We made the methods for items very generic. We can easily change the effect of any particular item by modifying its applyItemEffect method which will not require any changes in logic to other modules. Additionally, we subclass Item into Steppable and Usable Items, which implement onStepped() and onUse() respectively. The logic for when and how the player can pickup or use items may change but it will not require changes to these classes so long as player still calls onStepped() and onUse() on the items. Our item system has low coupling. To add more items we just subclass Steppable

or Usable Item and give it an applyItemEffect(). Additional triggers for items can also be added by subclassing Item.

- **Potions**: Since we apply the decorator pattern for potion effects, we can easily add more temporary values that are level-specific, since the temporary potion effects are cleared with a new floor.

Answers to Questions:

Note that the overarching approach for each question is the same as in Deadline 2, albeit with minor edits to reflect details of our ultimate implementation.

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?

We have created a Player class and have all of the races as subclasses (Human, Elf, Orc, Dwarf) of Player. Then by calling the default constructor for a race, we have a new Player. By default, this character will be a Human. A Player of any race is generated this way.

Adding additional races is also easy. In the default constructor for a race, we call the Player constructor with the appropriate stats. We then write an overriding method for whatever unique functionality that class might have. For example, if we add a race that attacks uniquely we just need to make attack() a virtual method and then write our new implementation in the subclass.

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

For reference, we represent the game board with a 2D vector of tiles. Tiles have a chamber number, which allows us to ensure that enemies cannot leave the chamber in which they begin. Entities and Tiles follow a subject-observer relationship so that Tiles will be notified when an Entity dies and can delete it, removing it from the game.

The GameModel is in charge of generating the enemies. At the beginning of the game, we create a 2D vector of Tiles from the given map layout by loading it in from a file. To generate an enemy, we select one of the subclasses of Enemy according to the relative probabilities of each Enemy class, pick a random chamber number, and then pick a random tile from that chamber. Then we instantiate the enemy and assign it to the appropriate tile.

Tiles reference Entities and Player is a subclass of Entity so this same generation method can be used for both Enemies and Players. The only difference is that only one instance of Player is created, whereas multiple enemies are created.

How could you implement special abilities for different enemies? For example, gold stealing for goblins, health regeneration for trolls, health stealing for vampires, etc.?

Abilities for enemies could be done by setting some functions as virtual and creating an override for that particular enemy as Enemies have subclasses for each Enemy type (Currently holding Dragon, Merchant, Phoenix, Troll, Goblin, Vampire, Werewolf). We can also use NVI to reuse

common aspects of the method. For example, to implement gold stealing for goblins and health stealing for vampires, for instance, we would use the afterAttacked() method, which is called in our attack method. Then, Goblin and Vampire can apply their special ability without changes to any other class.

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?

In order to track temporary potion effects, we use a decorator class called PotionFx. Each PotionFx object contains the change to atk or def associated with one potion that has been drunk. Whenever a player needs to know their current atk or def they will call getAtkChange() or getDefChange() on their PotionFx object to get the temporary changes. When the player goes to the next floor, we can simply delete their PotionFx in clearPotions().
Note that potions with permanent effects, including restore and poison health in our case, affect the player by mutating its HP value directly. This ensures that they are not impacted by the removal of temporary effects, since no PotionFx object is created for them.

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?

We have an abstract base class called Item that is an ancestor of all classes representing items in the game. As a result, the generation code can be reused from item to item, and since Entity is a superclass of Item, most of the logic can even be reused between item and character generation. The only unique step is calling the specific constructor for the desired item.

The code for protecting the dragon hoards and the Barrier Suit is reusable since we just need to have the BarrierSuit and the dragon hoards be locked by default and instantiate a dragon pointing to the locked items. The code is the same for both cases.

Extra Credit Features:

Memory Management: We are using unique_ptr to handle memory management implicitly. We are not using shared_ptr as it is slower, more complicated, and not relevant to our use cases since our objects do not have multiple owners. The places in which we require pointers for ownership are:

- In Game, which owns a GameModel and a InputDevice
- In GameModel, which owns a 2D array of Tiles
- In Tile, which optionally owns an Entity
- In Player, which optionally owns a Potionfx
- In Potionfx, which also optionally owns a Potionfx

However, raw pointers are still used in places such as our observer pattern implementation, since these pointers do not represent ownership.

Phoenix: Phoenixes are able to respawn after they die. In particular, if a phoenix has regenerated less than three times, its loot is another phoenix. Otherwise, it drops either a compass or has no drop as applicable. Each regeneration has progressively less health than the last.

Vampire: If the player kills a vampire, then as it dies, it will reveal which type of enemy holds the compass on the current floor. This allows for the game to be more balanced as it limits the number of enemies that the player must kill to obtain the compass. This feature necessitated incorporating the observer pattern with the GameModel class, allowing GameModel to elegantly add text to the printout at the end of the player turn.

Goblin: If Player has a compass, then the Goblin will steal the compass after it is attacked by the Player. To do this, we had to add an afterAttacked method within the Enemy superclass to produce post-attack effects. This method also allows our merchant to turn hostile once attacked. We also had to add setters to the private compassAcquired field of Player and a hideStairs function for the Tile class since we had only had getters for picking up a compass.

Hp Shrine: A health shrine, exclusively on the second floor, exchanges 1.0 gold for 20 hp. This process required creating a new Usable Item which can only be used if the player has enough gold. The rationale behind this extension was to account for the difficulty of the first level and to also give the player something to do with their gold.

Stats Shrine: A stats shrine, exclusively on the second floor, exchanges 10 hp for either 20 atk or 20 def, with a 50% chance for each option. This required us to make our attack and defense stats non-const and calling die once a player's health falls below 0 as a punishment for gluttony. This required creating a new Usable Item for the purpose of awarding well-playing players.

Final Questions:

1. **What lessons did this project teach you about developing software in teams?**

Planning for Collaborative Design: Although planning beforehand initially seemed like an obstacle since we all thought of ourselves as plan-as-you-go coders, planning a structured UML diagram before coding truly helped us to work on separate sections and still be on the same page. We were all pleasantly surprised with the minimal merge conflicts and conflicting ideas in general. Whiteboarding also helped with this endeavour.

Clear Definition of Roles: This follows from the first point, but something as easy as "What are you working on? Oh, I'll work on this then" prevents two people from overwriting each others' code. This is vital when multiple people work on interconnected features. Clearly defining roles also ensured that everyone was on-task so that the workload wasn't split in a skewed manner.

In-person Work Sessions: The best decision we made was to meet in person at a designated spot to work on this project for an entire afternoon/evening. This helped effective and efficient communication since we could chat about decisions as we were coding, fix bugs together, and look at each other's code if something was wrong. Rubber ducking helped save lots of time in debugging errors originating from the idiosyncrasies of C++, or from hard-to-spot corner cases. We were also able to take short breaks together and stay focused due to the productive environment we created.

2. **What would you have done differently if you had the chance to start over?**

Take More Time Planning: In general, we did a good job of planning out potential complexities during the design process. However, we realize now that the majority of our deviations from the initial design came from a relatively small proportion of the program specifications, which means that our failure to properly consider niche edge cases led to notable changes later on to prevent coupling or allow for the transfer of data between two parts of the program. Although most of these were simply adding new accessors and mutators, more drastic alterations could have been necessary. Looking back, to prevent these annoying changes, we should have taken more time to truly consider the many edge cases and more thoroughly examine the specification before we started programming.

Be More Proactive: We did not start planning our design until a couple days prior to Due Date 2 as a result of other coursework, and we paid for this with many stressful hours of trying to figure out whether Tile should have a pointer to Entity or vice versa. In addition, once we reached a conclusion, we would read the program specification to find an edge case that would convince us otherwise. In hindsight, we agree that this could have been prevented with a more proactive approach. This experience taught us to start coding the day after we submitted the plan of attack to maximize our time planning and coding, as well as simultaneously minimizing our stress levels. Thus, we felt well-prepared and not stressed as the final deadline approached.