1. UML Diagram. (see another document)
2. A list of design patterns and why/how, including the names of all classes involved in implementing the pattern.

(1) Abstract Factory design pattern. There are lots of subclasses of the Entity class with clear classification. Also, they interact with each other by dealing damage, transacting items, physical colliding, etc. In order to draw their graphics obscurely and easily, we follow the Abstract Factory pattern to implement all files in a graphics package. To do this, we created 10 drawers (factories) for these families: bullet, door, **enemy, gun**, merchant, **player**, health bar, level, **presenters, and room.** The bolded families have more than one class, so the drawer decides which constructor to call.

(2) Observer design pattern. The enemy class depends on the player class, and we want the enemy to attack the player when the player comes to the enemy's room. In order to do this, we want players to be able to notify the enemy in the Observer design pattern. The involved classes are Enemy and Player in game.entities.abstractions package. The enemy is the observer and the player is observable. When the player notifies the enemy, the enemy sets the target for the player.

(3) Strategy pattern. Used when an IGraphGenerator is passed into the level. Also used when IDoorDrawer, IPlayerDrawer, IGunDrawer, etc. are passed into Door, Player, Gun, etc. respectively.

(4) Singleton pattern. Used to hold and return error messages and codes. The implementation is done through the use of *enum*, with an implicit private constructor which prevents the creation of multiple instances, as prescribed by the singleton pattern.

(5) MVC pattern is used for the login system, with adherence to clean architecture: dependency is always maintained in the correct direction. The model entities are User and Log, and the use-case business logic class is User Controller. Between them lies the User Manager, which is how the two layers interface. AppUI, acting as the view, uses these classes, none of which ever calls AppUI.

3. A list of design decisions and explanations about how your code has improved since Phase 1.

(a) Moved out data constants into a config file to isolate hardcoded values from entity classes and simplify the management of values related to game configurations.

(b) Error codes and corresponding error messages are moved out into a separate enumeration class Msg for better type safety during error handling and greater flexibility for future multilingual support.

(c) **Removed the inventory manager.** Initially, we let the Player and Merchant store InventoryManager deal with item collection/transaction/equipment. However, it makes entities dependent on controllers which violate the

dependency rule. So, we replaced the one in merchant with an ArrayList variable itemOwned, and removed the storage of items in Player. The player can only change armor/use a health flask at the merchant. Also, we reimplemented the item operations at the entities level. Therefore, the entities depend on other entities, preserving clean architecture.

(d) Used dependency injection in merchant/manager, and fixed the hard dependencies in the room class, thus improving encapsulation.

(e) **Reorganized all files.** We divide the configuration, login system, entities, graphics, use cases, and controllers into well-named packages. Furthermore, we partitioned files into more packages by their roles in the maze game. It makes the project structure easy to follow.

(f) Added restrictions to the Player. Now, the player has less "power" than before. For example, when a player does transactions with a merchant, the merchant will decide whether the player has enough gold (money). Also, the player can collect an item only when it collides with it (i.e. close enough).

(g) **Made the Point class variable x,y private.** Before refactoring, the x, and y variables of point have public access modifiers, and they can be accessed/modified. Although it provides an ease-to-use point class, it may expose entities' location. In order to encapsulate it, we made x, and y private and created a getter/setter.

(h) **Refactored merchant.** Now, the merchant depends only on the player class. The input controller takes in commands related to interacting with a merchant and passes them through Room to the CurrentRoom's merchant, who then handles the logic, such as checking to see if the player is proximate and has enough gold and moving items between inventories.

4. How your program does or COULD follow the 7 principles of universal design

(1) Principle 1: Equitable Use.

(a) In the beginning, all players have 100 healths, 100 golds, the same weapon, and no armor. All item and enemy spawn points are completely random. In addition, the common goal of all players is to defeat the enemy. Therefore, the game ensures equality for everyone.

(b) The login system manages the player's username and password. Others cannot log in to the player's account and access account information. Therefore, the player's privacy is protected.

(2) Principle 2: Flexibility in Use

(a) Although the rules are the same, there are many ways to complete the game. According to the user's pace, they can choose to kill the boss and enemies at first or kill them after collecting gold and getting the strongest equipment.

(b) The login system supports both admin users and normal users. The admin can suspend the malicious users. Also, parents can create an

admin account to view the minor's logger history. Both provide flexibility in use.

(3) Principle 3: Simple and Intuitive Use
    (a) The moving forward/left/back/right operation can be done by keys W/A/S/D or ▲/◄/▼/►. Also, during transactions with a merchant, the n-th numeric key corresponds to the n-th item in the merchant's inventory. The keyboard operation is very intuitive, and even the first-time player can guess what operations can be done by the keys.

(4) Principle 4: Perceptible Information
    (a) The most essential information is the user's health, so our program generates a large red health bar. When the player is dead, the screen displays the legible words "YOU DIED". The less important information, such as shield, gold, and gun, are displayed on two sides of the screen.
    (b) the information of the merchant's items will only be displayed on collision, so it won't block the player's sight when the player does not want to buy the item.

(5) Principle 5: Tolerance for Error
    (a) When the player's gold coins are not enough to buy the item, the system will display the purchase failed and cancel the transaction.

(6) Principle 6: Low Physical Effort
    (a) This is a real-time game requiring fast reflexes, which is not for everyone, but we still make an effort to follow this principle. The game is operated by a mouse and keyboard. Most operations such as gun firing and buying can be done by a single keyboard/mouse click. No operation except moving requires holding a key, and no single operation requires clicking multiple keys, which minimizes physical effort.

(7) Principle 7: Size and Space for Approach and Use
    (a) In the drawer classes (e.g. SandRoomDrawer), the screen width and length can be changed to fit the screen or to meet visual needs.