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.

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

(a) **Removed the inventory manager.** Initially, we let the entity store inventoryManager to deal with item collection/transaction/equipment. However, it makes entities dependent on controllers which violate the dependency rule. So, we replaced it with an ArrayList variable itemOwned, and we reimplement the item operations at entities level. Therefore, the entities depend on other entities, which reserve clean architecture.

(b) used dependency injection in merchant/manager,

(c) fixed the hard dependencies in the room,

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

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

(f) **Made the Point class variable x,y private.** Before refactoring, the x,y variables of point have public access modifier, and they can be accessed/modified. Although it provides ease to use point class, it may

expose entities' location. In order to capsulate it, we made x,y private and created getter/setter.

(g) **Refactored merchant.** Now, the merchant does not depend on any other entities. The input controller directly handles transactions with merchant by clicking keys.

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, no 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 sight when the player does not want to buy 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) 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.