Task 1

Part (a)

i.) Mercenary.java and Zombie.java both use similar code in **move**:

```
            // Move random
            Random randGen = new Random();
            List<Position> pos = getPosition().getCardinallyAdjacentPositions();
            pos = pos
                .stream()
                .filter(p -> map.canMoveTo(this, p)).collect(Collectors.toList());
            if (pos.size() == 0) {
                nextPos = getPosition();
            } else {
                nextPos = pos.get(randGen.nextInt(pos.size()));
            }
```

```
        Position nextPos;
        List<Position> pos = getPosition().getCardinallyAdjacentPositions();
        pos = pos
            .stream()
            .filter(p -> map.canMoveTo(this, p)).collect(Collectors.toList());
        if (pos.size() == 0) {
            nextPos = getPosition();
        } else {
            nextPos = pos.get(randGen.nextInt(pos.size()));
        }
        map.moveTo(this, nextPos);
```

ii.) The Template Method can improve the quality of the code and avoid repetition, by extracting this random movement code to define a ‘default’ random movement in the superclass Enemy.java, which both Mercenary.java and Zombie.java can use. However, Mercenary.java can override steps without changing the default movement’s structure, as Mercenaries may also move towards the player if hostile.

iii.) Took the random movement code and implemented in the superclass, which both Mercenary.java and ZombieToast.java call. However, Mercenary.java also can override the random movement and choose to move towards the player, should they be hostile.

Part (b)

GameMap.java's lines 202 to 205, which describes the method **destroyEntity**(Entity entity) is an example of the Observer Pattern. This is because the subject (GameMap) has an observer/subscriber (Game), and when GameMap experiences a state change, which in this case is an entity 'dying', it notifies Game automatically by calling **onDestroy**.

Part (c)

i.) The most notable violations of the Law of Demeter is in the code relating to Entities.

- In Enemy.java, the method **onOverlap**(GameMap map) gets the object Game from GameMap and then calls a method of Game (**battle**) ✔
- In Enemy.java, the method **onDestroy**(GameMap map) gets the object Game from GameMap and then calls a method of Game (**unsubscribe**) ✔
- In ZombieToastSpawner.java, the method **onDestroy**(GameMap map) gets the object Game from GameMap and then calls a method of Game (**unsubscribe**) ✔
- In ZombieToasterSpawner.java, spawn call **game.getEntityFactory().spawnZombie(game, this);** ✔
- In Spider.java, the method **move**(Game game) gets the object GameMap from Game and then calls a method of GameMap (**moveTo**), on line 62. ✔
- In ZombieToast.java, the method **move**(Game game) gets the object GameMap from Game and then calls a method of GameMap (**moveTo**), on line 33. ✔
- In ZombieToastSpawner.java, the method **interact**(Player player, Game game) calls **player.getInventory().getWeapon().use(game).** ✔
- In BattleFacade.java, the method **battle**(Game game, Player player, Enemy enemy) calls:
    - player.getBattleStatistics().getHealth(); ✔
    - enemy.getBattleStatistics().getHealth(); ✔
    - BattleItem item : player.getInventory().getEntities(BattleItem.class) ✔
    - player.getBattleStatistics().setHealth(playerBattleStatistics.getHealth()); ✔
    - enemy.getBattleStatistics().setHealth(enemyBattleStatistics.getHealth()); ✔
    - if (!playerBattleStatistics.isEnabled() || !enemyBaseStatistics.isEnabled()) ✔
    - playerBattleStatistics = player.getBattleStatistics(); ✔
    - enemyBaseStatistics = enemy.getBattleStatistics(); ✔
- In Bow.java and Shield.java, **game.getPlayer().remove(this);** ✔
- In Player.java, **map.getGame().battle(this, (Enemy) entity);** ✔

Current changes made:

- GameMap.java has functions **doDestroy** and **doBattle**, which Enemy.java and ZombieToastSpawner.java call instead
- **move** for Enemy.java now has the parameter GameMap map instead of Game game
- **interact** for ZombieToastSpawner.java instead calls a method **playerUseWeapon** in Player.java
- ZombieToasterSpawner.java calls function **doSpawnZombie**, a method in Game.java.
- BattleFacade.java now instead calls a function **getHP** in Player.java and in Enemy.java.
- BattleFacade.java now calls a function **getItems** in Player.java.
- BattleFacade.java now calls a function **changeHP** in Player.java and Enemy.java.
- Enemy.java has a function **checkEnabled** to return a boolean of **isEnabled**()
- Player.java has a function **checkEnabledBuffed** to return a boolean **isEnabled**() for the buffed version
- Game.java has a method **destroyItem** which removes an item from a player's inventory.

Part (d)

i) The design here is poor, because it uses multiple if else statements to consider the different states while the states are already implemented. The logic for the individual states are all unique to themselves, so it would make more sense to push that logic into their own individual classes. Also, creating booleans for isInvisible and isInvincible is not a good design, as we can utilise the classes themselves to distinguish between which states the player is in.

ii) The code was refactored by removed the isInvincible and isInvisible methods. Then, the code in Player was adjusted so that it only needs to call an applyBuff() method from state to change the state. The booleans isInvisible and isInvincible were removed as well.

Part (e)

i) The code smell present in the code is that the three overridden methods are not necessary as they don't do anything. This is an unnecessary abstraction.
There was the same code smell in:

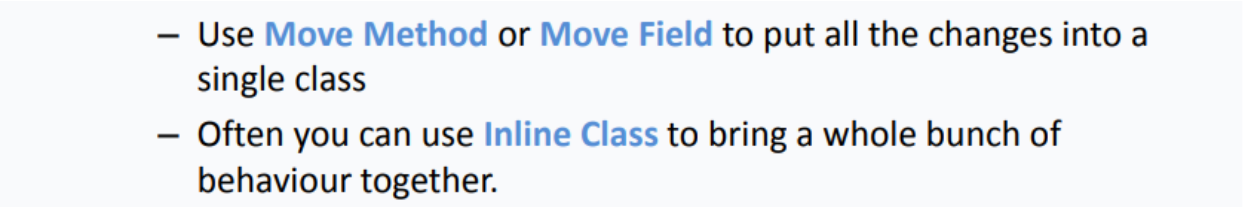
- Boulder.java
- Door.java
- Player.java
- Portal.java
- Switch.java
- Wall.java

ii) Instead of Entity being an abstract class, I changed it to a class and changed the abstract methods to normal methods. I also removed all the unnecessary overrides.

Part (f)

i.) The design smell noticeable is ‘Shotgun Surgery’. That is, a small change in the code forces a lot of little changes to different classes, which in this case made the previous engineering team abandon the changes.

ii.) As taken from the notes, the solution to ‘Shotgun Surgery’ is to:



Seemingly, the main cause of this ‘Shotgun Surgery’ smell is that Entity define methods such as onOverlap, which means if we want to override this method for collectables, we have to individually override it for each collectable. This is because we can’t define change onOverlap in Entity.java as that might make players be able to pick up enemies, which is a bit silly…

Essentially, I made a subclass of Entity.java, called Collectable.java that collectable items extend. The method of **onOverlap** in Collectable.java will override Entity.java. Since the player ‘overlaps’ onto the collectable entity, Collectable.java’s onOverlap calls doPickUp, a method in Player.java, so that pick up of items is now done in the Player class.

Note that it was important to create the subclass Collectable, as it lets us put all the differences of collectable entities vs enemies/others in a single class. I also removed some pointless return methods in the subclasses of Collectables (such as Treasure.java).

Part (g)

i) The design is not of good quality, as it relies on a switch statement for type every single function inside goals. This does not follow the open-closed principle, as we have to make changes to the function every time we add another game state. The design should be changed to include states, so if new states were to be added, it will be much easier to manage.

ii) GoalFactory has an okay design, as it needs the multiple cases to parse the json to create a google. A state pattern would be more preferable inside Goal, as all the game states include the same functions. An abstract class of Goal can be used to solve this, so each individual class can have their own required attributes and the code can be much more easily managed.

Part (h)

- In Responsebuilder,java, when constructing a dungeon response, there is an unnecessary check for whether the goal is achieved or not, as it is done within the classes themselves

Task 2

Part (a)

i)

Requirements:

- Create a new goal to ensure a certain number of enemies are killed, along with all spawners
- Assuming number of enemies is larger than 0

Design:

- Create a new subclass of Goals, called EnemyGoal.java, which will contain the desired amount of enemies required. ToString and achieved will need to be implemented.

Test List:

- Testing one enemy
- Testing three enemies
- Testing one enemy and one spawner
- Testing three enemies and two spawners
- Testing treasure with exit with one enemy

Changelog:

- Changed the code up in Game.java, so that the zombie spawners are actually removed properly, so they actually get removed.

ii)

Requirements:

- Allies that aren't adjacent to players will follow the player
- When the ally is adjacent to the player, they should occupy the last square the player was in
- They should NOT move inside the player

Design:

- Implement behaviour for allies, so that they follow players via Dijkstra algorithm
- Remember the player's last square so the ally can occupy it
- Multiple allies CAN occupy the same space

Test List:

- Test one enemy becomes ally
    - Test the following behaviour
- Test one enemy becomes ally, other stays enemy
- Test two enemies become allies
- Test enemy becomes ally next to player

iii)

Requirements:

- Assassins are mercenaries that do more damage
- They have a chance to fail bribes

Design:

- Implement a new class Assassin.java
- Change EntityFactory as well, since you need to create assassins
    - Make a new method for creating/returning assassins

Test List:

- One assassin fight player (and kills player)
- One assassin bribe success
- One assassin bribe fail

Part (b)

i) a)

Requirements:

- New collectable item called sun stone
- It can open doors
- Acts as treasure
- Acts as a key
- Cannot be used to bribe mercenaries or assassins
- Never gets destroyed, even after opening doors or building entities

Design:

- Create a new subclass SunStone.java
- Check up on how bribing, opening doors and craftings items will work

Test List:

- Counts towards treasure goal
- Can open regular doors and is retained
- Cannot open switch doors (save for switch doors)
- Can be used to craft and is retained
- Cannot bribe mercenaries or assassins

Assumptions:

- Crafting will prioritize using sun stones as opposed to other materials
- Opening doors will prioritize sun stones over using the keys

Changelog:

- Created a SunStone class
- Updated graghNodeFactory and EntityFactory to include sun stones
- Updated checkBuildCriteria() for Inventory.java to include sun stones in crafting
- Changed the treasure goal so that the sun stones are now included in the count
- Changed the initial treasure count
- Updated getBuildables in Inventory.java

- Updated door.java to allow for sun stones

i) b)

Requirements:

- Create a new item sceptre, which can mind control mercenaries/assassins at any range
- However, only lasts for so many ticks

Design:

- Create new class Spectre.java, a buildable
- Change the method of crafting items to implement crafting sceptre
- Change interact to also check if player has a sceptre

Test List:

- Test craft Sceptre and MC mercenary
- Test craft Sceptre and MC assassin
    - Test that MC breaks on mercenary
    - Test that MC breaks on assassin
- Mind control the same enemy twice
    - Can't mind control them while still mind controlled

Requirements:

- Create a new item Midnight Armour. Midnight armour provides extra attack damage as well as protection, and it lasts forever.
- Requires Sword + Sun Stone

Design:

- Create new class MidnightArmour.java, a buildable
- Change the method of crafting items to implement crafting midnight armour
- Official name is midnight_armour.

Test List:

- Test craft Midnight armour
    - Compare damage with vs without midnight armour
    - Compare damage taken with vs without midnight armour
- Test infinite durability of midnight armour

ii)

Requirements:

- New tile called swamp tile that is fixed in place
- Slows entities, except for player and allies right next to the player
- Swamp tiles each have their own movement factor, which is the number of ticks it will be stuck on the tile for

Design:

- New class called Swamp.java that acts as an entity
- It will contain a movement attribute that it will be acting on with others
- Other entities will have a swampTick attribute, that will be continuously added until it exceeds the movement factor of the swamp tile they are on

Test List:

- Test 0 movement
- Mercenary should be slowed
- Zombie and assassin is slowed
- Test player not slowed
- Ally next to player is not slowed
- Ally not next to player is slowed\

Changelog

- Added isAdjacent() to mercenary
- Changed mercenary movement to check if they are allied and are (or on the last tick) adjacent to the player
- Added movement factor attribute inside mercenary
- Added moveOnto override
- Changed entity and node factories to include swamp tiles
- Added swamp factor in the tile and the graph node
- Calculations are done in the move command, as it only affects moving entities
- Added check for a swamp tile before move in classes

iii) Persistence

Requirements:

- At any point during a game, the game should be capable of:
    - Saving to a local persistence layer, so that if the application is terminated, the user can:
        - Reboot the application
        - Select the game from a list of saved games
        - Continue playing from where they left off
- Need to preserve the position of entities on the map
- Need to preserve persistence of potions, bribing/mind-control etc.

Design:

- Make a new class for saved games, which can be stored in a list
- Each saved game should preserve time, positions and item persistence
- Modify **DungeonManiaController** and the methods for saving/loading games
- Modify **GameSave**

Test List:

- Test to show that positions are preserved
- Test to show that positions & time are preserved
- Test to show positions & time & items are preserved
- Test with multiple saved games

Currently saving configs/dungeons in:

- src/main/resources/saved_configs/
- src/main/resources/saved_dungeons/

iv) a) Time Turner

Requirements:

- User can collect a time turner
- Can travel back one tick, or 5 ticks
- Older self needs to exist in rewinded map
- Inventory persists
- Older self is an enemy
- Enemies follow new player and not old
- Time travel only occurs once here
- Older player can interact with other objects

Design:

- Make a new time turner collectable
- Use save game to save game states and load them for the ticks
- Player now needs a way to become an enemy

Test List:

- Test to show a time turner is picked up
- Test positions are correct for time travelling
- Test older self exists
- Test mercenary movement
- Test old player interacting with objects

Changelog:

- Added a time turner entity and changed node/entity factor
- Loaded game from a certain tick, by saving the game after every tick
- To fix the double player issue, addPlayer is a method added to game.java.
- The player is fetched from the current game, before the game from the previous tick is loaded
- setOld is a method and attribute added to player, to determine if they are the old player or not
- The player in game.java would then point to the current player, as opposed to the player that is loaded in from the tick
- Added a queue for the positions that the old player needs to move to, to ensure the old player moves in the correct position
- isRemovable() is added to player, where it checks if the player is old and the player has moved to all the squares it needs to. When it is true, that means the time travelled entity is done, so it can be removed from the map.

v) Logical Entities

Requirements:

- User can use boulders to activate switches
- Switches that are activated, activate adjacent entities
- Wires that are activated also activate adjacent entities
- Light bulbs are initially unactivated and can be activated if they’re logical is met

- Switch doors are similar, they can be entered if their logical is met
- logicals:
    - AND, can be activated if there are two or more adjacent activated conductors, if there are more than 2 conductors adjacent then all must be activated
    - OR, can be activated if there are one or more adjacent conductors active
    - XOR, can only be activated if exactly one adjacent conductor is active
    - CO_AND, will only be activated if two or more adjacent conductors are activated in the same tick

Design:

- Make a new wire entity
- Make a new switch door entity
- Make a new light switch on entity
- Make a new light switch off entity
- Whenever a boulder overlaps with a switch on the map, check adjacent cells on the maps for conductors that can be activated and repeat with those conductors
- When a light bulb is turned on, remove the off light bulb and replace it with an on one
- Check the number of conductors and active conductors adjacent to light bulb or switch door and see if it satisfies the logical, if it is CO_AND also check the tick of the game

Test List:

- Test when a switch is right next to a light bulb with OR
- Test when a switch is right next to a switch door with OR
- Test when a wire connects a switch and a light bulb
- Test when a wire connects a switch and a switch door
- Test when a boulder is pushed off
- Testing AND for a light bulb
- Testing AND for a switch door
- Testing XOR for a light bulb
- Testing XOR for a switch door
- Testing CO_AND for a switch door
- Testing CO_AND failing for a switch door
- Testing CO_AND for a light bulb

Task 3

One part of the codebase which does not adhere to the actual code requirements, is the zombie spawners not being properly removed, when they get interacted with. Even though they do not spawn zombies, they should be removed from the board. Hence, the test inside Zombie test now expects 0 zombie spawners after the interaction.

Tags

Test	Tag Number
Movement	1
Interface	2
Boulders	3

Doors Keys	4
Buildables	5
Potions	6
Portals	7
Bomb	8
Spiders	9
Zombies	10
Battle	11
Mercenary	12
Basic Goals	13
Complex Goals	14
Assassin	15
Sun Stone	16
Sceptre	17
Midnight Armour	18
Swamp	19
Persistence	20
Time Turner	21
Logic Switch	22