Task 1) Code Analysis and Refactoring ⛏️

a) From DRY to Design Patterns

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/1

> i. Look inside src/main/java/dungeonmania/entities/enemies. Where can you notice an instance of repeated code? Note down the particular offending lines/methods/fields.

In Mercenary.java from line 103 to 129 is the same movement method as in ZombieToast.java from line 27 to 53. The method “move” is repeated.

> ii. What Design Pattern could be used to improve the quality of the code and avoid repetition? Justify your choice by relating the scenario to the key characteristics of your chosen Design Pattern.

Initial thoughts: I think since movement is an important method with a lot of variations perhaps we put it in its own class? And use something like strategy design pattern to switch between different movements? Make a super class of movement where subclasses can inherit and add in their variations. All movement related methods and attributes should be in the superclass.

Characteristics of strategy pattern:

- Lets you indirectly alter the object’s behaviour at runtime by associating it with different sub-objects which can perform specific sub-tasks in different ways.
    - Which means that it suits our case since we want to use different variants of movement algorithms.
    - Also we have a massive conditional statement that switches between different variants of the same algorithm such as in Mercenary class. Strategy pattern can be used to extract all algorithms into separate classes, all of which implement the same interface.

> iii. Using your chosen Design Pattern, refactor the code to remove the repetition.

Instead of having all the movement behaviours in their specific enemy classes, I made a movement folder and made movement specific classes since some entities might have similar movements such as in our case and then we can just easily refer to that specific movement class or strategy instead of rewriting it all over again.

b) Observer Pattern

> Identify one place where the Observer Pattern is present in the codebase, and outline how the implementation relates to the key characteristics of the Observer Pattern.

The player object of entity class and the subclasses of Potion is an example of the observer pattern where player (observer) receives updates from InvincibilityPotion and InvisibilityPotion about the buff through “applyBuff” method. In this case, the potion subclasses are subjects because they maintain state (the buff) that can change, and they are responsible for notifying he player when this state changes. The player object is the observer as it receives updates about the buffs from the potions. The player object implements the “applyBuff” method, which allows it to react to changes accordingly.

c) Inheritance Design

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/3

> i. Name the code smell present in the above code. Identify all subclasses of Entity which have similar code smells that point towards the same root cause.

Exit entity inherits from entity class but does not override any of its methods. A bunch more have this problem where the methods are overridden with just the body “return”. This is a code smell because there is no point having these in the subclasses.

For example, the subclasses of entities arrow, key, treasure, and wood have methods that are not relevant to the object. These methods are onMovedAway and onDestroy are methods which are not applicable to these objects. Furthermore, onOverlap and canMoveOnto are repeated throughout all these subclasses. Hence, we should create a class which inherits from entity which contains methods: canMoveOnto and onOverlap. Arrow, key, treasure and wood classes should now inherit from this class.

In entity, we shouldn’t keep these 3 methods abstract methods. Remove abstract. Subclasses where these methods are relevant in, we can override these methods and leave out methods not needed.

> ii. Redesign the inheritance structure to solve the problem, in doing so remove the smells.

I removed abstract from these methods in the entity class so that these methods are all public void which returns nothing. In the subclasses, I removed the methods that returned nothing but kept the ones that override the methods.

d) More Code Smells

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/9

i. What design smell is present in the above description?

The code smell present is known as “Shotgun Surgery”. This happens when a single change in the codebase requires multiple small changes to be made in several different classes/files. This makes code difficult to maintain (as expressed in the developer statement) as changes have to be tracked across multiple locations. In this case, the attempt to change how item pickup is handled resulted in the need for changes in many different places in the codebase.

In all the collectable items, there are two methods common, which are canMoveOnto and onOverlap. When moving pickup from entity to player level, the engineers had to modify onOverlap method so that it works at player level. They had to do this for all the collectable items. Hence, making this task a hassle.

ii. Refactor the code to resolve the smell and underlying problem causing it.

I made a superclass called “Collectables” which inherits from entity and which the collectable items inherit from. This was to prevent DRY as the onOverlap method had repeated code for all the collectables. Hence, adding this method into a superclass “Collectables” makes it much easier to maintain and change in the future if something like this occurs again.

e) Open-Closed Goals

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/6

i. Do you think the design is of good quality here? Do you think it complies with the open-closed principle? Do you think the design should be changed?

The design is not of good quality. When looking at the code in Goal, the first code smell that is obvious is the extensive/complex use of switch statements which is discouraged in OOP. Generally, when you see a switch statement you should think of polymorphism. Furthermore, we can see that within the achieved and tostring methods, the same switch statement is used which violates the DRY principle. Lastly, it also violates the open-closed principle since this principle states that software entities should be open for extension but closed for modification. Looking at the goal class, we can see that adding a goal would require modifying both the achieved and tostring methods, as well as the GoalFactory class. This violates the open-closed principle and makes code harder to maintain. Due to all these problems above, the design should be changed. To improve it, we can apply the Strategy pattern.

ii. If you think the design is sufficient as it is, justify your decision. If you think the answer is no, pick a suitable Design Pattern that would improve the quality of the code and refactor the code accordingly.

Every goal was just hardcoded in the Goal class and it was messy, so I just used strategy pattern and moved them to their specific classes. Now, it just picks the strategy and the

implementation of the strategy is just in its respective class which makes for a open/closed design and therefore makes adding new goals easier.

f) Open Refactoring

Player States Refactor

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/7

The states are implemented using just boolean values to represent them, but that's really bad and hard to change and represent. So, I removed the booleans and refactored the states such as Invinciblestate and Invisiblestate in the triggernext and usepotion methods in player class to just check instance of potion and simply apply the state.

Entity Factory Refactor + smaller refactors

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/10

This was a big refactoring. Basically buildables had lots of hardcoding and entity factory had duplicated codes all over so I refactored them using factory pattern and created a main factory file as an interface and for every file needing to be created made a factory class, so the specific making specification of each file can be just in its own class and we don't need to copy code in entity factory. I also made other smaller refactors such as moving the getting available positions in entityfactory spawn spider to map class.

Buildables CheckbuildCriteria refactor

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/11

Moved functions of buildable into their specific classes, bow and shield. This way its easier to add more buildables and their specific build requirements, in preparation for task 2d.

Task 2) Evolution of Requirements 👽

a) Microevolution - Enemy Goal

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/12

Assumptions

- Any allies destroyed by a player-placed bomb count towards the enemy goal is undefined.

Design

In the game class, add a attribute called “enemiesKilled” which keeps in count how many enemies have been killed during the game. In the “init” method in this class, set the enemiesKilled value to 0. Furthermore, add a getter and setter for the number of enemies killed for use in other classes.

In goals package, add a new class “EnemyGoals” which inherits from “GoalStrategy” as we are implementing a new type of goal. The goal is achieved if number of enemies killed is greater or equal to the target amount of enemies specified. Furthermore, like the other goals, this class will have a toString method. Thus, within the GoalFactory class, we have to add another switch case for enemy goal.

Changes after review

The design seems good except that in onDestroy method in Enemy class, we need to set the enemiesKilled incrementing it by one each time enemy is destroyed.

Test list

Test achieving basic enemy goal: Like the previous other goals, I will implement a basic enemy goal similarly to the other ones previously implemented.

Test with one enemy goal: This is a test where the enemy kill goal is only 1 and the test implements player killing one spider enemy.

Test with both enemy goal and exit: This is a test where there are two goals. One enemy goal and one exit goal. In order to pass, the player must achieve both goals. I want to make it a little complex so I will be adding in 2 spider enemies in the dungeon and making enemy goal requirement 2.

Choice 1 (Bosses)

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/14

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/17

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/18

Assumptions

- The chance that the health of a Hydra increases when it gets attacked each round. The value of this field should be always inclusively between 0 and 1.
- The amount the health of a Hydra increases by when it gets attacked.

- The chance that the bribe on an assassin will fail. The value of this field should be always inclusively between 0 and 1.

Design

When creating the assassin, we need to write an assassinFactory class similarly to all other entities. When creating the assassin class itself, all the methods are the same to mercenary as they are extremely similar. The only differences is that assassin class will have more attributes such as bribe fail rate and a seed for random generator. In the assassin method, it will be similar to mercenary, only difference are the addition of extra attributes. Furthermore, in the interact method of assassin, we have to check whether random generator produces a number greater than the bribe fail rate. If it passes then the assassin is allied otherwise, it is not and assassin is hostile.

For Hydra, movement methods are same as zombie toast. Add in attributes for health increase rate and health increase amount. Add in onOverlap override function and check if health increase rate is greater than random, if so, add the increase amount to hydra health.

Changes after review

Add assassins case in entityFactory and graphNodeFactory. Everything else seems to be correct. And do the same for hydra.

Test list

The tests of assassin will be same with addition of a few new tests to mercenary tests. This includes:

- Test Assassin in line with Player moves towards them
- Test Assassin stops if they cannot move any closer to the player
- Test assassin can not move through closed doors
- Test assassin moves around a wall to get to the player
- Testing a assassin can be bribed with a certain amount
- Testing a assassin can be bribed within a radius
- Testing an allied assassin does not battle the player

The above tests are all same as mercenary as they are very similar entities. Below is the addition:

- Testing a failed bribe assassin does battle the player: tests whether assassin will battle player if it doesn't pass bribe fail rate. This one tests the sole purpose of this entity, and something that cannot be tested in mercenary.

For hydra,

- Test for simple movement: check if the movement of hydra is working
- Testing hydras cannot move through closed doors and walls

Other notes

[Any other notes]

Choice 2 (Sunstone & More Buildables)

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/13

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/15

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/M13C_IVYSAUR/assignment-ii/-/merge_requests/16

Assumptions

- Where there are multiple valid options for creating a buildable entity, the precedence of items is undefined
- The behaviour of a sceptre after use is undefined
- The behaviour of possessing multiple sceptres is undefined
- When trying to open a door with both a key and a sunstone in the player's inventory, it is undefined which entity will be used.
- The behaviour when `mind_control_duration` is <= 0 is undefined.
- When a mercenary or assassin can be bribed and mind controlled at the same time, which action will be taken after the player interacts with them is undefined.
- Whether midnight armour counts as a weapon when destroying zombie toast spawners is undefined.

Design

For this task, we need to implement two types of entities, one collectable and two buildables. Luckily we already refactored them and it will be easier to integrate new ones with the open/closed design. For the Sun Stone, we'll need to create a new class SunStone that extends from collectables and implements inventoryitem and also a SunStoneFactory for its creation. This class will just be a simple item you can pick up and the main functionalities and unique behaviours such as being used to open doors and being used in crafting will be implemented in their respective classes. Since it can be initiated from a map, we need to add it to GraphNodeFactory and modify EntityFactory. Also for door functionality, we need to add sunstone similar to a key. For the buildables, midnight armour is kind of similar to shield so just make a MidnightArmour class and extend buildable and similarly make Sceptre class and extend buildable.

Test list

Sunstone tests:

- Test player cant walk through a door without sunstone or key
- Test player cant walk through a door, but can use sunstone to unlock
- Test player cant walk through a closed door without key/sunstone
- Test player can build shield with sunstone

- Test player can build shield with sunstone then use that sunstone to open door

Buildables Tests:

- Test building midnight armour
- Test building sceptre

Other notes

Unfortunately, not enough time for more sceptre testing so not certain of functionality.

Task 3) Investigation Task ⁉️

After thoroughly reviewing the codebase and specification, I am confident that the software satisfies all the assigned requirements.

1. Comparison with the Requirements Specification

I started by comparing the MVP document and requirements with the current software. I verified that all the features and functions described in the specification were implemented in the software.

2. Functional Requirements Verification

Next, I verified the functional requirements of the software. I examined each function of the software to ensure it performs as expected. This was obviously easy because of proper testing throughout the implementation stage.

3. Non-functional Requirements Verification

I also checked the non-functional requirements. This includes checking the software's performance and usability. I tested with the interface to make sure the game runs properly and the frontend and backend work together.

4. Requirements Validation

Lastly, I validated the requirements with my teammate. This was done through a series of meetings and feedback sessions.

Through this process, I discovered a few areas where the software needed more test cases such as 2d. Unfortunately we didn’t have enough time to test for mercenary and assassin interactions with sceptre but if we had time we would’ve tested that feature.

In conclusion, while there were a few areas that needed improvement, the software mostly met the requirements.