Assignment II Pair Blog

Task 1) Code Analysis and Refactoring ⛏️

a) From DRY to Design Patterns

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/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.

There are no direct instances of repeated code inside Enemy.java. However there are many in its subclasses:

- Spider.java
    - updateNextPosition() contains repeated logic in if-statements which could be relocated to a helper method
    - updateNextPosition() is called twice inside the first if-statement in move()
- Mercenary.java
    - move() logic inside the InvincibilityPotion if-statement in this file is identical to the move() logic inside the InvincibilityPotion if-statement in ZombieToast.java
- ZombieToast.java
    - move() logic inside the InvincibilityPotion if-statement in this file is identical to the move() logic inside the InvincibilityPotion if-statement in Mercenary.java
- ZombieToastSpawner.java
    - onMoveAway() overrides its parent method, but has the exact same implementation as its parent method in Enemy.java

> 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.

Some repetition above can be resolved through use of helper methods and refactoring structure of if-statements, as well as removing unnecessary overrides. But, a strategy pattern can be used to:

- Reduce repetition
- Handle implementation of move() inside the InvincibilityPosition if-statement
- Handle other movement strategies such as the implementation of move() inside the InvisibilityPotion if-statement in Mercenary.java

This is especially useful in Enemy.java since (apart from the Player) enemies are the only entities that are movable. Implementing a "movable" interface with strategies as classes is a perfect application of the strategy pattern here.

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

1. Added the moveStrategy package within the enemies package, added MoveStrategy abstract class and MoveSpider, MoveZombieToast and MoveMercenary subclasses
2. Moved all the implementations of the move methods in each of the relevant enemy classes to their respective MoveStrategy subclasses, with MoveStrategy.java holding general enemy movement helper methods.
3. Replaced code inside the move implementations in each of the relevant enemy classes with instances of the MoveStrategy subclasses
4. Refactored moveWithInvincibility, which contained Large Class code smell and previously was duplicated code in both Mercenary.java and ZombieToast.java
5. Refactored other small sections of code in moveWithInvisibility (previously in Mercenary.java)

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 Observer Pattern is present in Switch.java and Bomb.java. It is implemented where switches are the subjects, and bombs are the observers.

The Observer pattern characteristically consists of two interfaces: the Subject Interface and the Observer Interface. The Subject Interface has a field containing an array of observers, and methods to add observers, remove observers and notify observers. The 'notify observers' method is usually a 'for each' loop that calls update() on each observer in its array, letting all observers know that the Subject's state has changed. The Observer Interface contains a method update(), which causes the observer to take action based on the change in state of the subject.

In this codebase, bombs detonate based on the state of the switch they are placed cardinally adjacent to. Thus, they must 'observe' the switch so that if the switch is or becomes active, the bomb detonates. There is not a Subject nor Observer interface, however their implementations are present within the files Switch.java and Bomb.java respectively. Switch.java has a list of bombs (observers), and implements the methods subscribe(), unsubscribe() and onOverlap(), and Bomb.java contains method notify(), qualifying both as forming an Observer Pattern in this codebase.

c) Inheritance Design

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/2

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.

The code smell is Refused Bequest, as Exit is a subclass of Entity but does not use all methods of Entity, evident from the void return statements in the code above. Thus, it also violates the Liskov Substitution Principle, as the subclass is not exactly substitutable for its superclass Entity as it contains ineffectual methods.

Other subclasses of Entity which have the same code smell include Wall.java and Buildable.java. These both have three of these 'void' overrides. All the entities below (basically all of them) have one or two 'void' overrides:

- Boulder.java

- Door.java
- Player.java
- Portal.java
- Switch.java
- Potion.java
- Arrow.java
- Bomb.java
- Key.java
- Sword.java
- Treasure.java
- Wood.java
- Enemy.java
- ZombieToastSpawner.java

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

1. The methods which do not implement every single subclass of Entity were turned into interfaces. This ended up being three interfaces for the methods onDestroy, onMovedAway and onOverlap. They were also removed from the Entity class.
2. If a subclass of Entity included one of these three methods, it now implemented it, and the void return methods were removed from the class.
3. GameMap.java was changed, so that before it called each of the methods, it would check (using instanceof) that the Entity implemented the interface before casting Entity to the type of Interface.

d) More Code Smells

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/3

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

The design smell is shotgun surgery, as the previous engineering team wanted to change one thing (the way collectables are picked up) and would have had to make lots of changes in lots of places.

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

The underlying problem is the repeated code as in the code block below, which is in every collectable entity.

```
@Override

    public void onOverlap(GameMap map, Entity entity) {

        if (entity instanceof Player) {

            if (!((Player) entity).pickUp(this))

                return;
            map.destroyEntity(this);

        }

    }
```

To refactor:

1. Since all collectables are inventory items, I changed the inheritance structure of collectables and inventory items. I changed InventoryItem to an abstract class which extends Entity, and each collectable extends InventoryItem.
2. From there, pull out OnOverlap from each class (except Bomb.java which has a slightly different method) and put into InventoryItem, so that the superclass implements the interface rather than each individual subclass.

Now any change made to the way that collectables are picked up can be handled in the overall InventoryItem class, rather than each collectable entity subclass.

e) Open-Closed Goals

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/4

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 great quality as it does not comply with the open-closed principle, which is that software should be open for extension and closed for modification. This is because every time a goal is added, the three switch statements in Goal.java and GoalFactory.java must be appended, so those files must be opened (and they should not be). Consequently, I think the design should be changed so that it adheres to the open-closed principle.

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.

I think the design is not sufficient as it is because there is a way for the code to adhere to the open-closed principle. By redesigning Goal.java using the factory design pattern, we can remove both switch statements, which will also remove the need to open Goal.java for modification if we add more goals.

To refactor:

1. Created a class for each type of Goal, eg BoulderGoal, which extends Goal.java. As a result I also changed the class for every different return type in GoalFactory.java as per the factory method.
2. I added two methods and getters inside Goal.java, which replaces the switch statements and enables subclasses to retrieve the information they need.

3. Each subclass of Goal overrides at least one method and is encapsulated inside its own class.

Thus for future additions, all that is needed is a new class extending Goal which overrides the appropriate methods, and to append GoalFactory.java.

f) Open Refactoring

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/5

- Fixed repeated code and large class in Inventory.java
- Fixed repeated code in various classes in buildable package by adding some generic methods

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/10

- Fixed issue with Player State pattern not being used effectively, by including the applyBuff method in each of the state implementations
- Fixed state pattern if statement in Player class, that determined which buff to apply

Merge Request 2a — due to (accidental) commit before merge, the following changes were also included in the previous commit:

- Checked the entire system and fixed any Law of Demeter violations present
    - This involved adding many methods to Player.java, Game.java and GameMap.java to aid in passing information between classes
- Fixed repeated code in Game.java

Task 2) Evolution of Requirements 👽

a) Microevolution - Enemy Goal

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/7

Assumptions

- Whether allies destroyed by a player-placed bomb count towards the enemy goal is undefined. (from Ed)

Design

- EnemyGoal.java - extends Goal.java as the goal representing the enemy goal
    - Holds the number of enemies to be killed
    - Returns whether enemies killed count has been reached AND if there are no spawners left on the map
- Track the enemies killed count similar to how treasure is counted - in Game.java and in Player.java, but call it in Enemy.java
- Track number of spawners using existing method getEntities in GameMap.java
- Add config files, where a new field “enemy_goal” is added with a default int of 1.
- Add dungeon json files, where the goal condition includes “enemies”.

Changes after review

Very well implemented. No issues found. The implementation was well designed due to the work done on refactoring goals and enemies packages, making this a lot easier.

Test list

- Test enemy goal on its own - basic
- Test treasure goal and enemy goal
- Test treasure goal and enemy goal but multiple of each
- Test enemy goal and exit goal - ensure exit goal passes last
- Test boulder goal or enemy goal

Other notes

Zombie toast spawners didn’t disappear from the map at first, so we changed this. More info in Task 3 below.

Choice 1 (Snakes)

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/8

Main implementation of snake merged into master.

Assumptions

- A Snake can be separated into two classes: the SnakeHead and SnakeBody
- Snakes always pathfind to nearest item it can eat, then hibernate once no more items on the map
- Snakes can eat multiple items, with their buffs stacking (except for potions: this behaviour is undefined)
- Snakes can path through anything except walls and other snakes. This means it can path through boulders, portals, switches, doors, etc. without interaction with them
- Destroying a SnakeHead or a SnakeBody will both add 1 toward the enemy goal
    - This behaviour is undefined. Though now that I think about it, the consequence is that players would be encouraged to destroy all body parts of a snake before its head (and before it hibernates!)
- Invisible snakes cannot be attacked by players, but they can still pathfind to treasures
- We can use the list iterator to prevent concurrent modification error

Design

Files created for implementation of the Snake:

- enemies package:
  - SnakeHead.java (class) extends Enemy
  - SnakeBody.java (class) extends Enemy
  - moveStrategy package:
    - MoveSnake.java (class) extends MoveStrategy
- SnakeFood.java (abstract class) extends InventoryItem

Design

Many other files were changed too:

- EntityFactory.java: create a new SnakeHead, SnakeBody, and SnakeHead with same attributes of an existing SnakeBody
- GraphNodeFactory.java: updated to pass snakehead config info to EntityFactory.java
- Game.java: handles snake defeat cases after battle
- Wall.java: handles snake pathfinding cases depending on its state (invisible snakes can path through walls)
- Relevant classes that implement InventoryItem: fixed to now implement SnakeFood and include an override method to determine which buff to apply to the snake
- Relevant classes with an onOverlap override method: updated to handle behaviours with a snake
- A whole series of config and dungeon json files for testing purposes
- Two dungeon and one config json file for testing on the frontend
- resources/…/entities package: snake_head.png and snake_body.png added for frontend visualisation
- resources/skins/default.json: added png files for visualisation of snake_head and snake_body objects in the frontend

Changes after review

- Inheritance design looks fine, snake design is good and makes sense

Test list

- Test snake hibernating (a wild sleepy boi in its natural habitat)
- Test snake moving to single item (hereon referred to as food)
- Test snake moving to nearest food when there are multiple
- Test snake eating food growing its body
- Test snake body following its head after creation
- Test player destroys snake head in battle
- Test player destroys snake body in battle and all following body parts destroyed
- Test player is destroyed by overpowered snake
- Test player cannot attack hibernating snake
- Test snake receives arrow buff
- Test snake receives treasure buff
- Test snake receives key buff
- Test snake can path through walls while invisible
- Test player cannot attack invisible snake
- Test snake body parts form new snake while invincible when player attacks body

Other notes

onOverlap behaviour was so annoying to deal with

Choice 2 (Logic Switches)

MR:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/13

Assumptions

- That lightbulbs can also turn off when the logical condition is not satisfied anymore
- That lightbulb on and off are two different entities, and we delete one or the other depending on whether the lightbulb condition has been met or not
- That there will only be either logic bombs or non-logic bombs in a map
- If the circuit is destroyed as it is activated, whether entities are activated or not

Design

- The whole design implements the observer pattern - where each entity is watching its neighbours
- Three interfaces, all of which cover different levels of being a 'logical entity':
- Subscriber - gets notified by neighbouring Conductors, includes Wire, Logical Bomb, Light Bulb on and off and Switch Door
- Conductor - conducts current and notifies neighbouring Subscribers of changes, includes Switch and Wire
- Logical Entity extends Subscriber - these are subscribers who do not conduct current but activate depending on their logic field
- Entities:
    - LightbulbOn.java
    - LightbulbOff.java
        - The lightbulbs replace each other and delete themselves when activated or not
    - Wire.java
        - The wire is both a conductor and subscriber, as it watches other wires and switches and notifies wires and logical entities.
    - SwitchDoor.java
    - LogicBomb.java
        - Separate to bomb, has a few of the same methods (could be abstracted out) but otherwise different.
- LogicWrapper - a class that holds a logical entity, and performs methods to do with logical requirements, especially CO_AND, where it checks each entity's neighbours for when they were activated.

Changes after review

Small concern regarding repeatability of code inside Wire.java, but nothing major. Could be easily refactored. Tests look very exhaustive. Well done!

Test list

- OR lightbulb switches on and off according to condition
- OR switch door opens and closes according to condition
- AND lightbulb
- XOR switch door
- CO_AND lightbulb
- OR bomb
- AND bomb
- XOR bomb
- CO_AND bomb
- A simple circuit

Other notes

- After a bomb detonates, behaviour is a bit cooked but some of it is undefined.

Task 3) Investigation Task ⁉️

MR 1:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/6

The MVP spec includes “The Player can destroy a zombie spawner if they have a weapon and are cardinally adjacent to the spawner”. We found that zombie spawners, though ‘destroyed’, did not disappear from the map, nor stop spawning zombies. We have assumed that by ‘destroyed’, the spec means the zombie spawners should disappear from the map and as a result stop spawning zombies too. This was discovered during task 2a (microevolution of enemies) as zombie spawners need to be destroyed to achieve the enemy goal.

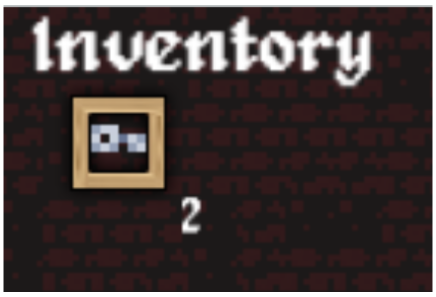
Changes made:

- Added a line in ZombieToastSpawner.java to destroy the entity after a battleItem is used on it
- Changed a test to ensure the spawner disappears after interacting with it

MR 2:

https://nw-syd-gitlab.cseunsw.tech/COMP2511/23T3/teams/T09A_CATERPIE/assignment-ii/-/merge_requests/9

MVP spec says players can only hold one key at a time. The implementation says otherwise.



In fact, a test explicitly approves of the inventory having two keys when the name of the test says “test cannot have two keys at the same time.”

Changes made:

- Changed Key.java so the onOverlap method handles if a player already has a key in the inventory
- Changed a test to ensure this behaviour is valid)