Assignment II Pair Blog

Assignment II Pair Blog

Task 1) Code Analysis and Refactoring ⛏️

a) From DRY to Design Patterns

Link to merge request

> 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 both the Mercenary and ZombieToast classes, there's a movement behavior when the enemy has an InvincibilityPotion. This portion of the movement logic is identical in both classes violating the DRY principle. It also heavily violates the open/closed principle as when each new movement needs to be added we have to write all the logic within the if/else statement

> 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.

The Strategy Pattern can be a good fit here. The strategy pattern is used when we have multiple algorithms (or strategies) for a specific task, and we can select one of these algorithms depending on the situation. Here, the movement behavior of enemies can be seen as a strategy that can change. Relating the code to the strategy design we observe that different movement behaviors of enemies are the algorithms. The behavior can be encapsulated in separate classes (strategies). Depending on the situation (like the type of potion the player has), a suitable movement behavior can be chosen.

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

The first step was to start implementing the strategy pattern. This involved making an interface with the common method nextPosition. For each type of movement, a strategy for it was created. The Enemy class was altered so that it had the ability to change movement strategies. Each Enemy subclass then replaced the movement logic with a change in strategy method.

The move class in Enemy was altered to take into account how the strategies would work and the value they would return, actually moving the enemy.

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 evident in the relationship between the Bomb class and the Switch class. The key characteristics of the observer pattern that are found in this is example are:

Subject: The Switch class acts as the Subject. It maintains a list of its dependents, in this case, bombs, and notifies them of any state changes.

Observers: The Bomb class acts as the Observer. It provides a mechanism (subscribe(Switch s)) to get added to the list of subjects to be notified. It also defines a notify(GameMap map) method that gets called to notify the bomb of any state change.

Subscribing and Unsubscribing: Bombs can subscribe to a switch using the subscribe(Bomb b) method in the Switch class.

Similarly, bombs can be unsubscribed using the unsubscribe(Bomb b) method in the Switch class.

Notification: When a Boulder overlaps with a Switch (onOverlap method in the Switch class), all subscribed bombs are notified, causing them to explode.

The notify(GameMap map) method in the Bomb class is the mechanism by which the bomb gets updated about any state change in the Switch.

The Switch and Bomb classes are loosely coupled. The Bomb doesn't need to continuously check the state of the Switch; instead, it gets notified whenever there's a change in the state of the Switch. This is a clear implementation of the Observer pattern.

c) Inheritance Design

Links to your merge requests

> 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 present in the above extract is refused bequest as those methods are not implemented and used in any meaningful way (methods return immediately). The methods onOverlap, onMovedAway and onDestroy are abstract methods in the class Entity that all subclasses of Entity are required to use. This causes this code smell when many of those subclasses do not need to use them in any meaningful way. Duplicate code is also a code smell as repetition of this occurs across all classes that directly extend abstract class Entity.

All subclasses of Entity that have similar code smells that point towards the same root cause include:

3 methods

- Wall
- Buildable
- Potion

2 methods

- Arrow
- Bomb
- Key
- Sword
- Treasure

- Wood
- Door
- Boulder
- Player
- Portal
- ZombieToastSpawner

1 method

- Enemy
- Switch

]

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

[The strategy was to delete the overriding methods that effectively do nothing in all subclasses of abstract class Entity. Next, the abstract methods within abstract class Entity were changed so they were normal methods that did nothing (simply returned immediately). This made it non-compulsory to implement overriding methods in all subclasses. To make using the methods optional for subclasses of Entity, those methods were moved into interfaces that could be implemented when necessary.]

d) More Code Smells

Link to merge request

Link to merge request

> What design smell is present in the above description?

The code smell present in the above description is a duplicate code which leads to the Shotgun Surgery smell. The smell is evident when it says in the note that when they tried to make changes they "had to start making changes in heaps of different places for it to work" signifying the presence of duplicated code since if we wished to change the pickup logic we have to change the code in multiple collectables

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

[Briefly explain what you did]

- I added the repeated code of the pickup logic into the interface InventoryItem. This was the pickup method for all items. I had it return a boolean so that individual classes, such as Bomb, that have class-specific logic surrounding the picking up of an item, can still have the logic depending on whether the item was picked up or not. This removed the duplicated code regarding pickup, allowing the pickup behaviour to be centralised in the InventoryItem interface.
- I then created a CollectibleEntity, an abstract class that extends an entity. This allowed the removal of duplicate onOverlap code across the vast majority of collectables and also the same overridden canMoveOnto method that was present in all collectibles. This removed the duplicated code smell and Shotgun Surgery problem.

e) Open-Closed Goals

Links to merge request

> 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 quality is poor because the code within the folder “goals” does not comply with the open-closed principle. If new features for goals were to be added, significant modification would have been added to Goal.java. In order to have existing code open for extension and closed for modification, the code within the folder “goals” must be refactored using an effective design pattern.

Another code smell present within the methods “achieved” and “toString” in Goals.java contain long switch statements, which is bad style and has poor readability. This should not be present after changing the design.

]

> 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.

No, the design is not sufficient. I decided to refactor the code to use a strategy design pattern. This resolves the violation of the open-closed principle and removes the long switch statements, making the code more readable. To do this, the logic from the switch statements were moved into newly created classes that characterise the strategy design pattern.

f) Open Refactoring

Merge Request 1

Issue: State pattern implementation for potions regarding the player state

Solution: Changed the way the state pattern was implemented. No longer violates open closed principle. In PlayerState, applyPotion calls applyEffect which changes the state of which potion is in use. The applyEffect method makes a new instance of whatever state it needs to transition into and sets this as the player's current state. This decouples the classes and makes it easier to add a new potion, having to only add a somepotionState and somepotion, some potion being any new potion/effect.

Merge Request 2

Issue: The current implementation of buildable entities contains a significant amount of hard coding. The use() and getDurability() methods were both duplicated in Bow and Shield violating the DRY principle.

Solution: Move all repeated methods and variables to superclass Buildable. Create variables for constants used in applyBuff() to remove hardcoding.

Merge Request 3 - Several Law of Demeter violations

Issue: Law of Demeter violation in ZombieToastSpawner.java within the method interact.

```
public void interact(Game game) {
    player.getInventory().getWeapon().use(game);
}
```

Solution: In ZombieToastSpawner.java, interact method is now [player.useWeapon(game);]. Two new methods were added:

- Player.java

```
public void useWeapon(Game game) {
    getInventory().useWeapon(game);
}
```

- Inventory.java

```
public void useWeapon(Game game) {
    getWeapon().use(game);
}
```

Issue: Law of Demeter violation in ZombieToastSpawner.java within in the method spawn.

```
public void spawn(Game game) {
    game.getEntityFactory().spawnZombie(game, zombieToastSpawner);
}
```

Solution: In ZombieToastSpawner.java, spawn is now:

```
public void spawn(Game game) {
    game.spawnZombie(game, this);
}
```

One method has been added to accommodate for this change in Game.java:

```
public void spawnZombie(Game game, ZombieToastSpawner zombieToastSpawner) {
    getEntityFactory().spawnZombie(game, zombieToastSpawner);
}
```

Issue: Law of Demeter violation in BattleFacade.java regarding the initialising and updating of player and enemy health.

```
// 0. init
double initialPlayerHealth = player.getBattleStatistics().getHealth();
double initialEnemyHealth = enemy.getBattleStatistics().getHealth();

// ...

for (BattleItem item : player.getInventory().getEntities(BattleItem.class)) {

// ...

// 3. update health to the actual statistics
player.getBattleStatistics().setHealth(playerBattleStatistics.getHealth());
enemy.getBattleStatistics().setHealth(enemyBattleStatistics.getHealth());
```

Solution: Changed BattleFacade.java to the following:

```
// 0. init
double initialPlayerHealth = player.getPlayerHealth();
double initialEnemyHealth = enemy.getEnemyHealth();

// ...

for (BattleItem item : player.getBattleItems()) {

// ...

// 3. update health to the actual statistics
player.setPlayerHealth(playerBattleStatistics.getHealth());
enemy.setEmemyHealth(enemyBattleStatistics.getHealth());
```

Added the following to Player.java:

```
public double getPlayerHealth() {
    return getBattleStatistics().getHealth();
}

public void setPlayerHealth(double newHealth) {
    getBattleStatistics().setHealth(newHealth);
}

// ...

public List<BattleItem> getBattleItems() {
    return getInventory().getEntities(BattleItem.class);
}
```

Added the following to Enemy.java:

```
public double getEnemyHealth() {
    return getBattleStatistics().getHealth();
}

public void setEmemyHealth(double newHealth) {
    getBattleStatistics().setHealth(newHealth);
}
```

Issue: Law of Demeter violations in Bow.java and Shield.java

```
game.getPlayer().remove(this);
```

Solution: Changed code in Bow.java and Shield.java to the following:

```
game.removeBuildable(this);
```

And added a new method in Player.java

```
public void removeBuildable(Buildable buildable) {
    getPlayer().remove(buildable);
}
```

More Law of Demeter violations that were fixed:

- Move method of Enemy.java && onOverlap method of Enemy.java (initiate battle)
    - New method moveEnemy in Game.java
    - New method battle in GameMap.java
- Move method of Mercenary.java
    - New methods getEffectivePotion in Game.java and GameMap.java
- nextPosition method in InvisibilityPotionMovementStrategy.java and DefaultZombieToastMovementStrategy.java
    - New method in Enemy.java
- nextPosition method in DefaultSpiderMovementStrategy
    - New method getNextPosition in Spider.java
    - New method getEntities in Game.java
- spawnSpider & spawnZombie method in EntityFactory.java
    - New method getPlayerPosition in GameMap.java
    - New method getCardinallyAdjacentPositions in ZombieToastSpawner.java

Task 2) Evolution of Requirements 👽

a) Microevolution - Enemy Goal

Link to Merge Request for Testing

Link to Merge Request for Implementation

Assumptions

- The ZombieToastSpawner is to be destroyed by interacting with it.
    - It can only be destroyed with a sword or bow
- The number of enemies needed to be killed to fulfil the goal must be at least 1. The exception is when a spawner exists on the map and the number of enemies killed is specified to be 0.

Design

- Methods/fields Added
    - Added the method getEnemiesKilled in Game.java to retrieve the number of enemies killed by the player
    - Added enemiesKilled field in Game.java
    - Added getEnemiesKilled() method to retrieve enemies killed
- Files Added
    - Added GoalEnemies.java to create the enemies goal in java
- Methods/Files modified
    - Modified the Game.java file as seen above
    - Modified GoalFactory to allow the creation of “enemies” goal.
    - Changed ZombieToastSpawner interact() to also destroy the entity
    - Changed battle method in Game.java to increment enemies killed when an enemy entity is destroyed

Changes after review

- A review was done and no changes were made to the initial design. We wanted to place enemiesKilled in player however this would only clutter the number of methods as there will only ever be one player in this version of the game.
- The second review revealed that the implementation of how zombieToastSpawner gets deleted is incorrect and must be changed by doing it through he interact mechanism.

Test list

- Test that goals are being set in the game
- Test basic enemy goal when an enemy exists
- Test basic enemy goal when spawner exists
- Test basic enemy goal when 2 enemies and 2 spawners exist
- Test complex goal (enemies AND exit)
- Test basic enemy goal when the number of enemies needed to kill is 0 and a spawner exists
- More complex goals???

Choice 1 (Sun Stone & More Buildables)

Links to merge requests

Assumptions

- A mercenary is mind-controlled by interacting with the mercenary when a sceptre is in the inventory of the player
- The sceptre can be used a maximum of three times before being destroyed

Design

- Methods/fields Added
  - Added canBeControlled and isMindControlled in mercenary
    - This is to check whether it is interactable
- Files Added
  - Added SunStone.java (as collectible), Sceptre.java and MidnightArmour.java (as buildable)
- Methods/Files modified
  - EntityFactory.java
    - Added method to build Sceptre and method to build Midnight armour
    - Added case to instantiate and place sun stone on the map.
  - Game.java

        - Edited build to check whether zombies are on the map to ensure midnight armour is not built if zombies are present
    - Inventory.java
        - Edited getBuildables to check if sceptre or Midnight Armour can be built
        - Edited checkBuildCriteria to:
            - Take in a string instead of a boolean for the type of buildable
            - Added checks for the shield to use a sunstone instead of treasure or key
            - Added checks for sceptre and midnight armour - removing the necessary items from the inventory and getting the entity factory to instantiate the buildable
    - Door.java
        - Modified to accept sunstone as a viable key
    - Player.java
        - method getSceptre() as a helper function.

Changes after review

- After a review, we chose not to make Sunstone a subclass of Treasure as it only made the implementation more complicated and hard to understand, even though it makes intuitive sense. It complicated the bribe logic mainly. Treating them completely separately makes the code much easier to understand and read.

Test list

Sunstones

- Sunstone can be picked up and placed in inventory
- Sunstone can open doors (kept after use)
- Sunstone can replace key/treasure in building (kept after use)
- Sunstone counts towards treasure goal
- Test bribe mercenary

Sceptre

- Test with all crafting recipes
- ‘bribes’ mercenary when interacted with sceptre in inventory
- Test for 3 ticks

Midnight Armour

- Test Crafting
- Test Crafting with zombie in dungeon
- Test attack damage buff and defence buff

Other notes

- Fixed bribe radius (wasn’t checked correctly to see if interactable).
- Fixed issue where player and mercenary share a square after a tick, instead of mercenary going to players old position.
- Fixed failing the mercenary test due to the wrong implementation of the bribe radius check
- Fixed implementation where BattleStatistics.applyBuff did not incorporate buff magnifier or reducer in new values, hence players would not receive 2x attack when using a bow
- Fixed test where the bow is used, ensuring correct health lost.

Choice 2 (Logic Switches)

Link to merge request

Assumptions

- When a switch is activated, conductance through all connected wires is immediate
- Players can stand on light bulbs
- Any scenario where the order in which activated components perform their action is undefined
    - For example, this case where a logical bomb might activate and destroy parts of a circuit before other logical components are able to activate

- If a switch activates bombs and wires at the same time, the bomb does not prevent current from running through a circuit.

Design

- Observer pattern:
    - Logical entities including logical bombs are observers to the subject wires
    - Wires are observers to the subject switch
- Inheritance relationships
    - LightBulb extends LogicalEntity
    - SwitchDoor extends LogicalEntity
    - logicalEntities folder in entities folder made for these classes
    - Bombs will implement similar methods of Logical Entity but not inherit form LogicalEntity (not in the folder either)
- GameMap (establish observer patterns)
    - Initialise relationships between wires (add wire connections)
    - Initialise relationships between switches and wires
    - Initialise relationships between wires and logical entities (light bulbs and switch doors)
    - Initialise relationships between wires and logical bombs
- Switch
    - Properties
        - Stores list of adjacent wires
    - Methods
        - Notify adjacent wires when switch is activated or deactivated
- Wire
    - Properties
        - Stores a list of adjacent switches

        - Stores list of adjacent wires
        - Stores a list of adjacent logical entities (light bulbs & switch doors)
        - Stores a list of logical bombs
    - Methods
        - Notify / update adjacent wires when wire becomes active with current
        - Notify / update adjacent logical entities when wire becomes active with current
- Logical Entity
    - Properties
        - String logic
        - bool all wires same state
        - Stores list of adjacent wires
        - Stores list of adjacent switches
    - Methods
        - checkLogic()
        - update() : void - returns immediately
        - Methods that add and remove from lists
- Lightbulb
    - Properties
        - boolean isOn (initialised to false)
    - Methods
        - update() overrides superclass method - turn on when logic is true and turn off when false
- Switch Door
    - Properties
        - boolean isOpen (initialised to false)

    - Methods
        - update() overrides superclass method - open when logic is true and close when false
        - onOverlap and onMovedAway methods need to be changed to activate wires and logical entities
- Bomb
    - Properties
        - Stores a list of adjacent wires
    - Methods
        - notifyLogic() - explode when logic is true and does not explode when logic is false
        - onOverlap() and onPutDown() methods needs to be changed to account for bombs being placed in a circuit
        - checkAllWiresSameState() to check “co_and” case
        - methods to add and remove wires
        - notifyLogic() added - called to check logic is met. If logic is met, explode bomb.
- Edit EntityFactory and GraphNodeFactory to initialise all relevant entities

Changes after review

- Edited NameConverter class to suit requirements
- New class Conductor extends Entity created - this involved some refactoring of code. This also resulted in an additional instance of an observer pattern where switch is the subject and adjacent logical entities to the instance of the switch are the observers. This was done due to the code smell of code repetition being present.
    - Wire extends Conductor
    - Switch extends Conductor

        - Both files placed in conductor folder in entities folder
        - More info
            - Properties
                - boolean activated
                - 3 Lists for wires, logical entities and logical bombs
            - Methods
                - adding and removing from lists
                - overriding canMoveOnto method that returns true
                - necessary getters and setters
- checkAllAdjWiresSameState() for “co_and” case
- Added tests 5 and 8 - 10 (shown below) later on to account for cases I missed
- Added more helper methods in Wire.java to deal with recursion and repetitive code I had initially.

Test list

1. Test AND logic with lightbulbs (switching on and off)
2. Test OR logic with lightbulbs (switching on and off)
3. Test XOR logic with lightbulbs (switching on and off)
4. Test CO_AND logic with lightbulbs (switching on and off)
5. Testing additional cases with regards to logic with lightbulbs
6. Test logic with switchdoors and test whether a player can walk through them when open and cannot walk through them when closed
7. Test behaviour of logical bombs in a circuit
8. Test case where bomb destroys a circuit and alters circuit behaviour
9. Test case where player picks up a logical bomb and places it in a circuit
10. Test case where switches and wires are involved in triggering logical entities

Other notes

- Added some code in existing methods to account for the extra functionality

Task 3) Investigation Task ⁉️

Through implementing new features and refactoring, we ensured that functionalities of pre-existing code were not changed and the MVP functionalities were present in the code. Allowing us to have a high confidence that the software satisfies its requirements

When implementing Task 2a the following was realised:

- The zombieToastSpawner was not able to be destroyed when interacting with it.
  - This was fixed by changing the implementation of its interact method to destroy the spawner
  - A faulty test was letting this happen and go undetected and was failing once correct implementation was done and so the test was fixed

When implementing Task 2d the following was realised:

- Fixed issue where player and mercenary can share a square after a tick, instead of mercenary going to players old position.
- Fixed failing the mercenary test due to the wrong implementation of the bribe radius check
- Fixed implementation where BattleStatistics.applyBuff did not incorporate buff magnifier or reducer in new values, hence players would not receive 2x attack when using a bow
- Fixed test where the bow is used and the check for health isn't correct.
- Fixed bribe radius check in mercenary by actually checking if player is in its radius

All the above changes were done in the Microevolution task merge request and in the Sun Stone and More Buildables merge request.

We also realised that bombs could have been refactored to utilise the state pattern however we did not have the time to implement this change.

Player can also move out of bounds.