SENG 301 Assignment 3 – Play Game

Harrison Cook – 52210542

Task 1 – Getting your hands dirty:

I created a new `Command` implementation called `EnterCommand`, the `execute()` function of which calls the `Character.enter()` function. In the `AdventureGame.play()` function, I changed the case for “leave” so that the body of the if-statement created a new `EnterCommand` with the character, set the `command` variable to the new enter command and then used `Action.setCommand()` to set the `action` variable’s command.

Task 2 – Collections:

To allow a user to collect multiple treasures of the same type, whilst retaining the collected order, I decided to use an `ArrayList`. My decision was made using the following considerations:

A Set would not have been appropriate here as they neither allow duplicates or guarantee order.

A List could’ve worked as they allow duplicates and guarantee order. The two List implementations I thought would be most appropriate were the `ArrayList` and the `LinkedList`.

A Map could’ve worked, such as a `LinkedHashMap` or a `TreeMap`, using either the `Room` object or visited room number the treasure was found in as the key (and the treasure as the value). Both of these allow duplicate (values) and guarantee some kind of order.

In the end, I decided to use a List vs a Map as they are a simpler Collection, there is no need to have to worry about creating `<key, value>` pairs for each piece of treasure the user collects. I decided to go for the simpler change, a List only needed two lines to be changed to function correctly, whereas a Map would’ve required many more changes. Since we needed to print out all the treasures the user collected, I decided to use an `ArrayList` over a `LinkedList` as we have no need to insert mid-way through the list or to delete the head of the list, which is where the `LinkedList` is beneficial over the `ArrayList`.

If we needed to keep track of the room the treasure was found in, a Map would be a better choice. Since we can use either the `Room` object or the room number the treasure was found in as the key, depending on our needs. A `LinkedHashMap` would be the best map option because it:

- Maintains insertion order
- Allows the Room object to be used as the key, or a room number (even if the rooms are visited out of creation order)
- Allows duplicate (value) entries – the user can collect multiple of each treasure

Task 3 – OO Wisdom:

Liskov Substitution Principle – The children of the `Room` class do not all adhere to the contract set out by the `Room` class. The `EnemyRoom`, `TreasureRoom` and `MixedRoom` classes require an `Enemy`, a `Treasure` or both, respectively, whilst the `Room` class itself requires none of these. This effectively is creating a new pre-condition: that `enemy` / `treasure` be instantiated before creating these classes (or null pointer exceptions may be thrown during runtime), this is tightening the pre-conditions which is not ok. This means the `EnemyRoom` / `TreasureRoom` / `MixedRoom` classes violate the principle: ***require no more, promise no less***. One way you could improve this is for the `Room` class to have the methods that generate the random enemies and treasures. This could be achieved by adding private enemy & treasure variables and moving the random generator methods into the `Room` class from the `AdventureGame` class. This would allow the child classes to utilise the random generator methods and would reduce coupling (a good thing).

Single Responsibility Principle – Currently, the `AdventureGame` class has multiple responsibilities: it both runs the game itself and sets up the board. As well as the previous suggestion, this can be solved by moving both the `setupBoard()` and `generateRandomRoom()` methods into the `Board` class. The `character.enter()` line can be moved into the `play()` method. The `setupBoard()` method could be made the `Board` class constructor taking an integer that represents `gameSize`.

Encapsulation Leak – In the `Character` class when we return the collection of treasures with the `getCollectedTreasures()` method, we should ideally be returning an unmodifiable collection, so the client(s) cannot modify the `collectedTreasures` collection. This is an easy fix, we can replace the line

```
return collectedTreasures;
```

with

```
return Collections.unmodifiableList(collectedTreasures);
```

which would solve this problem.

Information Hiding – Another flaw is that a lot of the `Board` class methods and variables are public. Since this program seems to be leaning towards getters & setters on the privacy continuum, the public variable should be private, and many of the public methods should be set to private. If the suggested changes are made above, the methods (and variables) that need to be public in the `Board` class, without making other changes are:

- `Board` constructor
- `getCurrentRoom()`
- `getBoard()`
- `changeRoom()`
- `endGame()`
- `isGameOver()`
    - If you call `allRoomsVisited()` from this method, otherwise `allRoomsVisited()` needs to be public as well. It seems to make more sense to have `isGameOver()` check it though as the game essentially ends when all rooms have been visited.

This is good as all of these are effectively getters and setters.

Stable Abstractions – The `MixedRoom` is an *unstable* abstraction, if we add more Room types we will likely want to add them to our `MixedRoom` class. This means `MixedRoom` is likely to change which makes it an *unstable* abstraction. You could solve this by abstracting out `MixedRoom`, possibly by using a Composite or Decorator pattern.

Task 4 – Make the board unique:

To achieve this, I followed the Singleton design pattern. I changed the `board` variable and the `Board` constructor to private. I then changed the `getBoard()` method to check if the `board` variable is null, if it is, create a new `Board` object and assign `board` to it.

Task 5 – Change Observer pattern implementation:

The originally implemented Observer pattern:

Gang of Four (GoF) Pattern: Observer	Implementation
Subject	`Character`
Observer	`Enemy`
Concrete Subject	`Character`
Concrete Observer	`Ninja, Turtle`
doSomething() / setState()	`fight(), kill(), prepare(), rest()*`
getState()	`isReady()`
attach() / detach()	`addEnemy()` / `removeEnemy()`
notify()	`updateEnemies()`
update()	`add()`
Subject – Observer relationship	`Character` – `Enemy` relationship
Concrete Subject – Concrete Observer relationship	`Character` – `Ninja` relationship `Character` – `Turtle` relationship

I decided to say that the `Character` class was both the (Base) Subject and ConcreteSubject. I said this because the (Base) Observer interacts with the `Character` class as if it was a (Base) Subject, and the ConcreteObserver classes interact with `Character` as if it was a ConcreteSubject.

* According to the GoF pattern, it seems as though a ConcreteObserver should be able to call the setState() (doSomething() in lectures) method (Gamma, Helm, Johnson, Vlissides, 1994, p. 295). Because of this, I have put down the four methods that the ConcreteObserver can call. However, the `changeState()` method is the method that conforms to setState() the best, so you could instead put that as the setState() method. In lectures, the setState() method was not elaborated on more than just saying it changes the state of the ConcreteSubject. Because of this I searched the internet and the original GoF Design Patterns book for information on the setState() method, which all seemed to imply setState() would need to be public, hence why I decided to put the four public methods that call `changeState()` instead of the `changeState()` method itself.

The newly implemented Observer pattern using Java's Observer:

GoF Pattern: Observer	**Implementation**
Subject	`Character`
Observer	`Enemy`
Concrete Subject	`Character`
Concrete Observer	`Ninja, Turtle`
doSomething() / setState()	`fight(), kill(), prepare(), rest()*`
getState()	`isReady()`
attach() / detach()	`addObserver()` / `deleteObserver()`
notify()	`notifyObservers()`
update()	`update()`
Subject – Observer relationship	`Character` – `Enemy` relationship
Concrete Subject – Concrete Observer relationship	`Character` – `Ninja` relationship `Character` – `Turtle` relationship

I decided to keep `addEnemy()` and `removeEnemy()` in the code (instead of using inherited methods everywhere they were called before) as it makes the program more readable, and reduced the number of changes made.

Task 6 – Recognising patterns:

Singleton (`Board` class):

GoF Pattern: Singleton	**Implementation**
Singleton	`Board`
uniqueInstance	`board`
instance()	`getBoard()`
Singleton()	`Board()`

Factory Method (`Enemy` creating `Weapon`):

GoF Pattern: Factory Method	**Implementation**
Creator	`Enemy`
ConcreteCreator	`Ninja, Turtle`
Product	`Weapon`
ConcreteProduct	`Hammer, Sword`
factoryMethod()	`createWeapon()`
doSomething()	`attack()`
ConcreteCreator – ConcreteProduct relationship	`Ninja – Sword` relationship `Turtle – Hammer` relationship

Command Pattern (`Command` class):

GoF Pattern: Command	Implementation
Invoker	`Action`
Client	`AdventureGame`
Command	`Command`
ConcreteCommand	`EnterCommand, LeaveCommand, SearchCommand`
Receiver	`Character`
execute()	`execute()`
unexecute	Does not exist
state	Does not exist
action()	`enter(), leave(), search()`
Client – ConcreteCommand relationship	`AdventureGame – EnterCommand` relationship, `AdventureGame – LeaveCommand` relationship, `AdventureGame – SearchCommand` relationship
Client – Receiver relationship	`AdventureGame – Character` relationship
ConcreteCommand – Receiver relationship	`EnterCommand – Character` relationship, `LeaveCommand – Character` relationship, `SearchCommand – Character` relationship
Invoker – Command relationship	`Action – Command` relationship

I decided to include unexecute here as it was presented in the lecture notes even though it is not included in the original GoF pattern.

Task 7 – Design pattern implementation:

The Composite pattern would add a large amount of value onto our original design, making the client's work simpler, removing an unstable abstraction, and making extension easier. Instead of repeating myself when saying why (in detail) Composition would add value and then talking about what principles were introduced / violated, I have discussed them together below.

Value added:

- ***Open / Closed Principle:*** Whilst the *Open / Closed Principle* was present in our design before, we have made it easier to extend our design. Adding the Composite pattern allows us to easily add new rooms into the design (and to mix and match those rooms to create different room combinations) and newly defined leaf-rooms or composite-rooms work automatically with existing structures (Gamma et al., 1994, p. 166).
- ***Simplicity (overarching principle):*** Adding the Composite pattern makes the client's work simpler, and the clients themselves simpler. Clients can treat composite structures and leaf objects as the same, and in fact don't need to know which they are dealing with. This simplifies client's code (Gamma et al., 1994, p. 166).
- ***Favour Composition Over Inheritance:*** Almost by definition the Composite pattern conforms to this principle, but by using this pattern we can create rooms with the characteristics of many other subclasses of `Room` without having to create those actual classes. By doing this we can keep our class hierarchy small and manageable.

- ***Stable Abstraction:*** By removing the `MixedRoom` class and 'replacing' it with the `CompositeRoom` class, we have removed the unstable abstraction from our program and replaced it with a stable one. `CompositeRoom` is stable because it does not need to be modified if more `Room` subtypes are added. I have not deleted the `MixedRoom` class in case it was wanted by the marker for reference, but I would do so normally.

Value removed:

- ***Interface Segregation Principle:*** This principle is violated by adding the Composite pattern to our design. We are forcing the client(s) to know about and depend on methods that they don't use and that aren't of interest to them. This means the client(s) know about methods that are not applicable to some of the objects they interact with.
- Whilst there isn't really a principle that has been weakened / violated, the Composite pattern has made the program worse and more confusing for the user. When iterating through the collection of rooms, because of how the program is written, many superfluous print statements are printed to the user. This could potentially be associated with the ***Simplicity*** principle(s) if you consider them to be related to user experience. This could be solved by letting the client(s) control when these statements are printed, instead of printing them when unrelated methods are called.
- One more issue with the Composite pattern in this implementation is although a `CompositeRoom` may contain multiple rooms with `Treasure`, only the `Treasure` from the first room encountered whilst iterating over the `room` collection is returned. This could be solved by changing the `Room` classes so they return a collection of treasures or by adding complex logic with flags to identify a `Room` as searched, and allowing to user to search a `Room` multiple times and retrieve multiple treasures. In this case I decided to opt for the simplest solution, with the least changes and just return the first `Treasure` found. This also applies to room with an `Enemy` as well.

Implemented Composite Pattern (`Room`):

GoF Pattern: Composite	**Implementation**
Client	`AdventureGame, Board`
Component	`Room`
Leaf	`BasicRoom, EnemyRoom, TreasureRoom`
Composite	`CompositeRoom`
doSomething()	`getEnemy(), getTreasure()`
add()	`add()`
remove()	`remove()`
getChild()	Does not exist
Client – Component relationship	`AdventureGame` – `Room` relationship `Board` – `Room` relationship
Composite – Component relationship	`CompositeRoom` – `Room` relationship

References:

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Mass: Addison-Wesley.