Nico Williams

011878369

[pdf format]

[class diagrams in star uml]

Creational(Abstract Factory):

1. Zombie abstract class serves a Product/ConcreteProduct
2. 4 classes inherit from zombie; RegularZombie, BucketZombie, ConeZombie, and ScreenDoorZombie
3. Abstract Factory class declares abstract methods for creating Zombies
4. ZombieFactories are place in concrete classes.
5. Client class is created for choosing a factory to create a zombie
6. ZombieFactory depend on Zombie class
7. Client class depend on ZombieFactory classes

composite:

Composite Pattern Diagram Explanation.

1.Zombie-Component

- takeDamage(int d)
- die()
- Interface
- The base for all concrete zombie classes

2.BasicZombie or RegularZombie

- Implements Zombie interface
  - takeDamage(int d)
  - die()

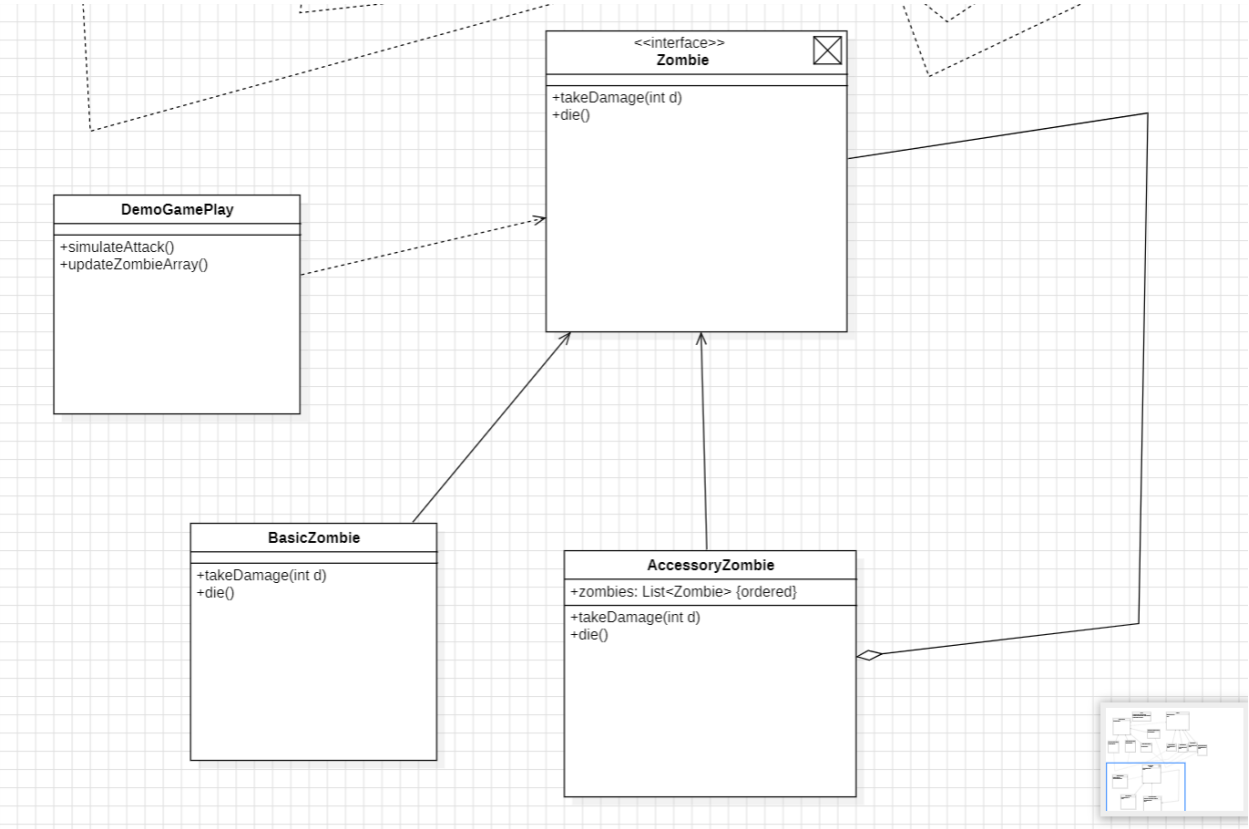
3.AccesoryZombie- Composite

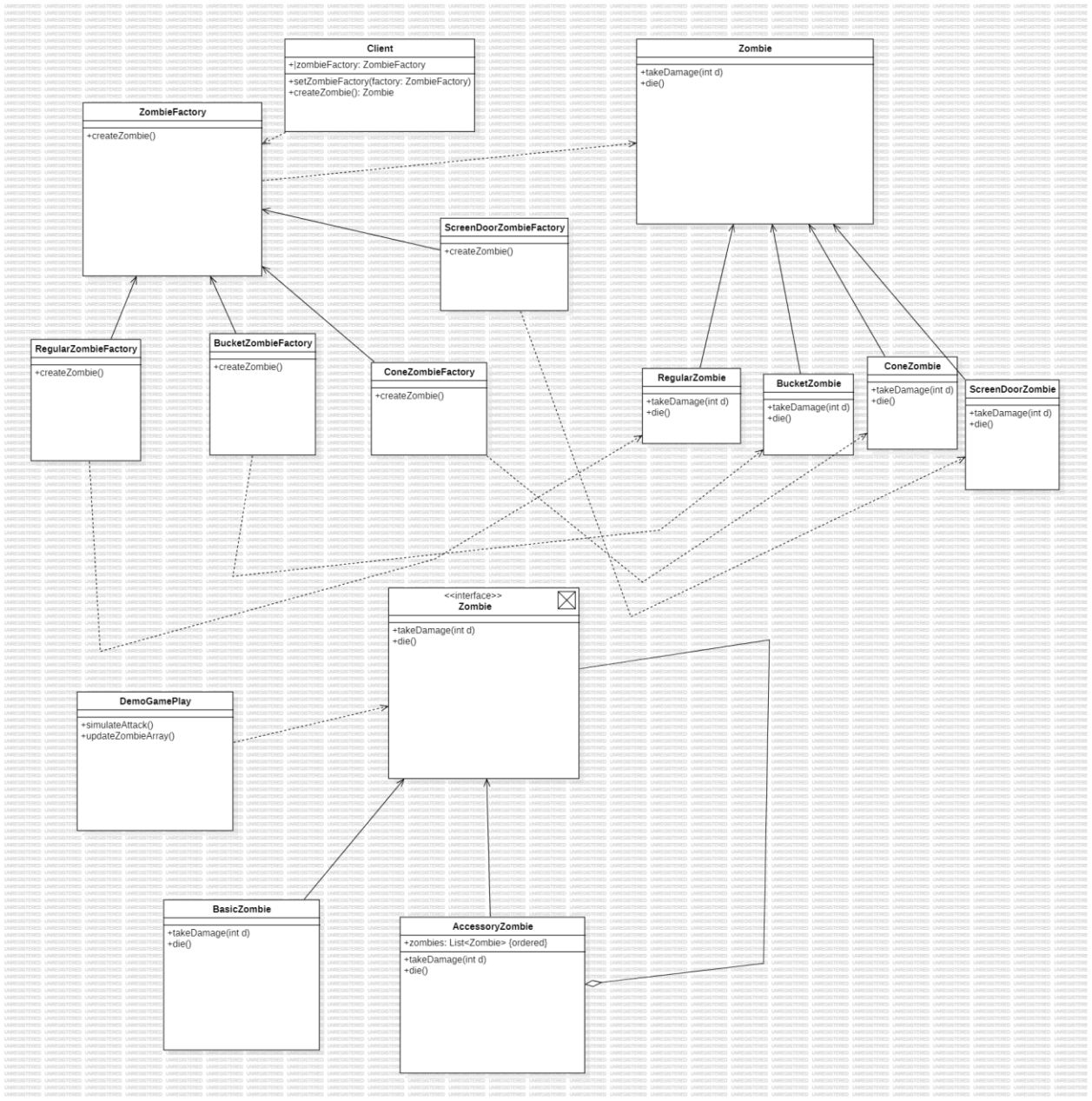
- Implements Zombie interface
  - takeDamage(int d)
  - die()
  - list of Zombie components
  - Represents Composite zombie with an accessory.

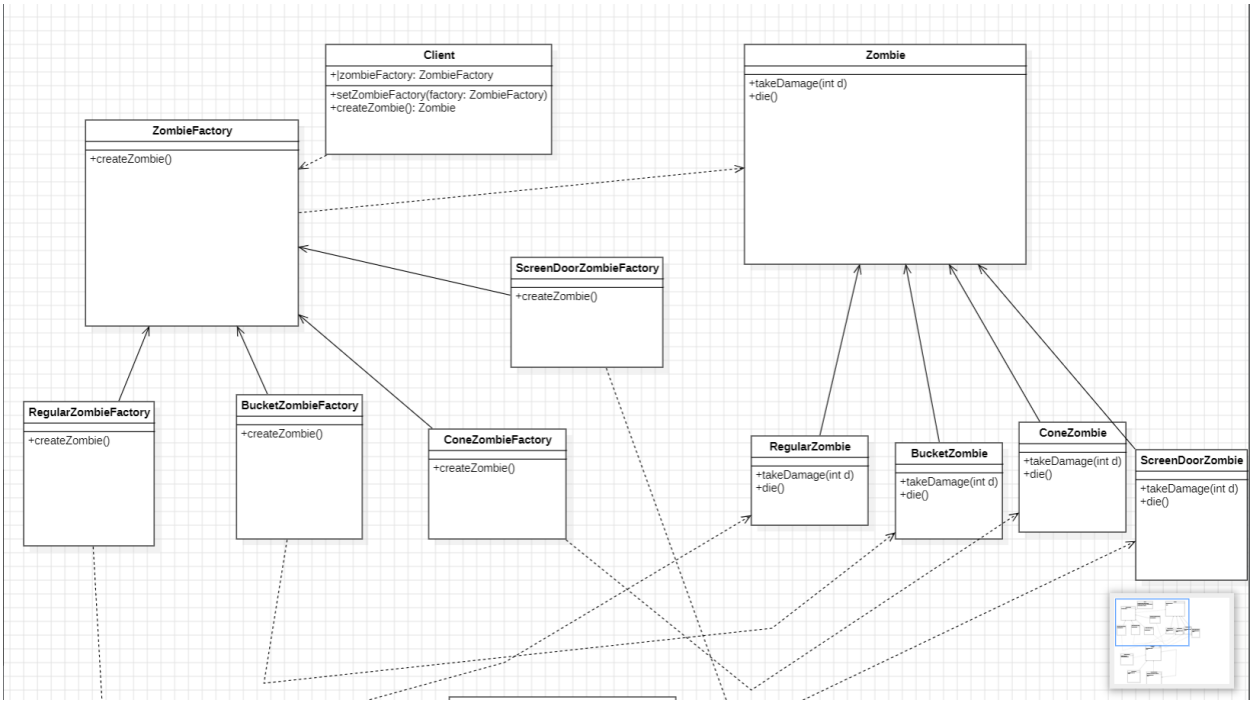
4.DemoGamePlay – Client

- List of zombie components
- Simulates attacks; uses takeDamage()

- Updates zombie status after attack


<<interface>>
Zombie
+takeDamage(int d)
+die()
DemoGamePlay
+simulateAttack()
+updateZombieArray()
BasicZombie
+takeDamage(int d)
+die()
AccessoryZombie
+zombies: List<Zombie> {ordered}
+takeDamage(int d)
+die()


Client
+|zombieFactory: ZombieFactory
+setZombieFactory(factory: ZombieFactory)
+createZombie(): Zombie
Zombie
+takeDamage(int d)
+die()
ZombieFactory
+createZombie()
ScreenDoorZombieFactory
+createZombie()
RegularZombieFactory
+createZombie()
BucketZombieFactory
+createZombie()
ConeZombieFactory
+createZombie()
RegularZombie
+takeDamage(int d)
+die()
BucketZombie
+takeDamage(int d)
+die()
ConeZombie
+takeDamage(int d)
+die()
ScreenDoorZombie
+takeDamage(int d)
+die()
<<interface>>
Zombie
+takeDamage(int d)
+die()
DemoGamePlay
+simulateAttack()
+updateZombieArray()
BasicZombie
+takeDamage(int d)
+die()
AccessoryZombie
+zombies: List<Zombie> {ordered}
+takeDamage(int d)
+die()



In context of the Abstract Factory and Composite patterns diagrams, associations may occur between classes such as “Zombie” and the Zombie Factory. (*regular zombie/basic zombie!*)

---

2. [10pts] Next to your diagram, list the match-ups between the classes in your diagram, and the ones shown in the pattern structure. For instance, “ZombieFactory” – Creator; “Zombie” – AbstractProduct; etc. This would be helpful for you to apply the patterns properly. To get full credits, you need to include the match-ups for both your choice of Creational pattern (Factory/Abstract Factory/Builder) and the Composite pattern.

Remember that when applying the design patterns, not only should the classes match up, the relationships (arrows between different classes), and the attributes/operations of the classes should match up too. Hint: Compare your answer with the slides, see if all the arrows are the right shape/direction, for instance.

Abstract Factory

1. ZombieFactory as a Composite Structure:

Match-up: The ZombieFactory acts as a composite structure that creates objects, similar to the Composite pattern creating composite objects. It creates different types of zombies, including both individual zombies (leaves) and composite zombies (containing accessories).

2. Derivation of AccessoryZombie from RegularZombie:

Match-up: The AccessoryZombie is derived from the RegularZombie. This relationship mirrors the hierarchy in the Composite pattern where composite objects (AccessoryZombie) may contain leaf objects "RegularZombie".

Composite Pattern

1. ZombieComponent as a Common Interface:

Match-up: The ZombieComponent class in the Composite pattern represents a common interface for both leaves "RegularZombie" and composites "AccessoryZombie". Similarly, the Zombie class in the Abstract Factory pattern acts as a common interface for different types of zombies.

2. Similarity in Manipulating Objects Uniformly:

Match-up: Both patterns promote a similar concept of manipulating objects and compositions of objects uniformly. In the Composite pattern, the client can treat individual objects and compositions uniformly, and in the Abstract Factory pattern, the client can create and manipulate different types of zombies uniformly through the common interface.

Match-ups

Composite Structures for Creating and Manipulating Objects

Match-up: Both patterns involve composite structures. The Abstract Factory pattern has a structure with factories for creating zombies, and the Composite pattern manipulates composite zombies containing accessory zombies.

Common Interfaces

Match-up: The Zombie class and ZombieComponent class act as common interfaces in both patterns, allowing clients to interact with complex structures uniformly.

3. [40pts] Write an executable demo program that follows your design above. The program should provide a simple command line interface of several commands. A sample command line interface may look like this:

a) [15pts] One option in the commands should allow the user to create different types of Zombies. For instance, when selecting “Create zombies”, the program should prompt the user which kind of zombie to create. The user may create several different zombies before stopping. All the zombies created should be saved in an array/vector.

After the user is done creating zombies, the user shall return to the main menu. The program shall display the array of Zombies that the user just created, with their type and health specified.

b) [25pts] Next, the user can simulate a plant (Peashooter) attacking the zombie by selecting the “Demo game play” option. For this assignment, we assume there is only one Peashooter.

The zombies are attacked one at a time from left to right. For instance, in the array above, the Regular Zombie will be attacked first until it dies, then the Cone Zombie will be attacked, and so forth.

Use the command line output to simulate the attack process: each time the plant attacks, print out an updated array of the Zombies and their remaining health values. Remember, the Zombies with an “accessory” should change type after the accessory is destroyed/removed, i.e. turn from a Cone Zombie to a Regular Zombie once the Cone’s health is gone, and this change should be reflected in your array display.

For example, these zombies:

After taking 25 damage three times, should be displayed as: [R/50, B/150, S/75], with the original first zombie dead and removed, and the second ConeZombie becoming a RegularZombie. See the screenshot below:

The demo should automatically continue until all the zombies are dead.

Hint: do not include a “Peashooter” class as there is no need. To simulate the attack, simply invoke the “takeDamage(int d)” function in a loop on your Zombies.

4. [20pts] What would change if we increased the damage of Peashooter to 40, instead of 25? Does this change impact your code and/or game logic at all? Write a short answer to this question, then accommodate this change into your program, allow the user to choose a different damage value by entering an integer after selecting “Demo game play” and before the game begins.

Then, implement “leftover damage” in your program by taking advantage of the Composite pattern: consider a Cone Zombie with full health 75; after taking a damage of 40, it should become a Regular Zombie with health 35 – NOT a full health Regular Zombie with health 50.

See the screenshots below as an example when damage value is set to 40.

Increasing the Peashhooter damage to 40 will require that the damage variable be updated and the takeDamage method in all the Zombie classes, become updated to account for the increased Peashooter damage to 40. (note* damage that exceeds an zombie health is rolled over to damage the next zombie once that zombie perishes, for example an attack killing off a screen door zombie, with more damage left will apply to the regular zombie, that the screen door zombie become when its health goes below the level sufficient enough to remain a screendoorzombie, and maintains its accessory(screendoor; cone, bucket).

5. [20pts] Say we want to introduce a new plant: Watermelon.

The way a Watermelon attacks is that it catapults a watermelon above and hit the Zombies from the top. Therefore, for Regular, Cone and Bucket Zombies, the Watermelon would function the same way as the Peashooter. However, for the Screen-Door Zombie, the Watermelon “goes above” the screen-door it’s holding, and hit the Zombie directly.

What does the new feature change about your program? Does the Composite pattern still work? DO NOT change your code for this question. However, write up a short explanation on how

this new change would impact your design and implementation. What kind of modification

would be necessary?

If we wanted to intoduce a new plant, Watermelon, we would have to;

- create a Watermelon class (thats similar to the Peashooter class)
- (since the watermelon attacks in a different way) The attack simulation methods and coding would need to be updated to allow the catapulting attack style (that goes above the screen-door and hits the ScreenDoorZombie).
- We must change the design code for the screen door zombies to accommodate the new watermelon attack in the code.
- Composite pattern still works and the watermelon can be added as new plant similar to peashooter, and maintaining the same composite design structure.